UNIVERSIDADE FEEVALE

SANDRO MADRUGA SILVEIRA

CRIAÇÃO DE UM FRAMEWORK MODULAR E EXPANSÍVEL PARA GERAÇÃO DE CÓDIGO FONTE

Novo Hamburgo
2012

SANDRO MADRUGA SILVEIRA

CRIAÇÃO DE UM FRAMEWORK MODULAR E EXPANSÍVEL PARA GERAÇÃO DE CÓDIGO FONTE

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Me. Edvar Bergmann Araujo

Novo Hamburgo
2012

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

A minha esposa Alice, aos meus amigos e colegas por entenderem minha ausência em alguns momentos em que estava me dedicando com toda motivação a este trabalho.

A minha família, sobretudo a minha mãe, falecida este ano, que sempre incentivou e se orgulhou das conquistas dos filhos nos estudos.

A Rech Informática Ltda, empresa da qual tenho orgulho de fazer parte, que apoia e subsidia a educação e formação acadêmica de seus colaboradores.

Também agradeço meu orientador Edvar Bergmann Araujo por todo o apoio durante a idealização e elaboração deste trabalho.

RESUMO

A utilização de ferramentas produtivas para o desenvolvimento de software garante o incremento de qualidade com redução de custos. Uma das formas de reduzir tarefas repetitivas durante a codificação é através da reutilização do conhecimento passado em novos projetos. Isto pode ser aplicado a partir de técnicas de geração automatizada de código fonte ao invés da reescrita de artefatos muito parecidos. Embora existam muitas ferramentas que se propõem a gerar pelo menos alguns trechos de código fonte, é bastante complexo atender situações específicas de cada projeto. Sendo assim, este trabalho tem como objetivo propor uma ferramenta de geração de código fonte flexível e que seja de aplicação simples em projetos distintos, onde a partir de algumas especificações básicas de entradas seja possível gerar vários tipos de artefatos de software. Também prevê que novas especificações possam ser implementadas ampliando as possibilidades de uso da ferramenta.

Palavras-chave: Geração de Código; Metadados; Padrões de Projeto; MDA (*Model-Driven Architecture*); DSL (*Domain-Specific Language*).

ABSTRACT

The use of productive tools for software development ensures the increase of quality with cost reduction. One way to reduce repetitive tasks while encoding is through the reuse of past knowledge in new projects. This can be applied techniques from automated generation of source code instead of rewriting artifacts very similar. Although there are many tools to code generate at least some parts of the source code is quite complex meet specific situations of each project. Therefore, this work propose a tool to generate source code that is flexible and simple to apply in different projects, where from some basic specifications for entries to be possible to generate various types of software artifacts. It also allows that new specifications can be implemented extending the possibilities of tool's use.

Keywords: Code Generation; Metadata; Design Patterns; MDA (Model-Driven Architecture); DSL (Domain-Specific Language).

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASP	*Active Server Pages*
BPM	*Business Process Modeling*
CSS	*Cascading Style Sheets*
DAO	*Data Access Object*
DDL	*Data Definition Language*
DLL	*Dynamic-Link Library*
DSL	*Domain-Specific Language*
EJB	*Enterprise JavaBeans*
EMF	*Eclipse Modeling Framework*
ERP	*Enterprise Resource Planning*
GUI	*Graphical User Interface*
HTML	*HyperText Markup Language*
I/O	*Input-Output*
IDE	*Integrated Development Environment*
ISO	*International Organization for Standardization*
J2EE	*Java Platform Enterprise Edition*
JDBC	*Java Database Connectivity*
JPA	*Java Persistence API*
MDA	*Model-Driven Architecture*
MDD	*Model-Driven Development*
MFC	*Microsoft Foundation Classes*
MVC	*Model–View–Controller*
OMG	*Object Management Group*
PHP	*Hypertext Preprocessor*
PIM	*Platform Independent Model*
POJO	*Plain Old Java Object*
PSM	*Platform Specific Model*
RPC	*Remote Procedure Call*
SQL	*Structured Query Language*
UI	*User Interface*

UML	*Unified Modeling Language*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*
XP	*eXtreme Programming*

SUMÁRIO

INTRODUÇÃO

A crescente demanda por soluções de software nas mais variadas áreas do conhecimento humano ampliou sobremaneira os requisitos, metodologias, técnicas e conhecimentos que os profissionais de engenharia de software precisam dominar. Toda essa exigência objetiva enfrentar de maneira eficaz e com alta qualidade aos vários desafios que se apresentam em todo o ciclo de vida de desenvolvimento. Há muito tempo entende-se que é necessário usar um amplo conjunto de ferramentas para o processo de desenvolvimento. Segundo Pressman (1995, p. 945): "Os engenheiros de software agora reconhecem que precisam de um número maior e mais variado de ferramentas (apenas ferramentas manuais não atendem às exigências dos modernos sistemas baseados em computador)".

Muitas abordagens são utilizadas no processo de desenvolvimento de software visando obter maior produtividade e qualidade. Tais abordagens são aplicadas desde a obtenção preliminar de requisitos até a manutenção corretiva e evolutiva de um software. No entanto, por mais maturidade que o processo de produção de software possa atingir, sempre existem projetos onde alguma parte será construída de forma artesanal. Comparado aos modelos industriais onde rígidos padrões e altíssimos níveis de automatização são usados, muitos problemas no desenvolvimento de software poderiam ser evitados, reduzindo custo e tempo necessários para a obtenção dos resultados.

Analistas de sistemas e programadores, de forma recorrente, criam soluções similares, muitas vezes aplicando conceitos baseados em algum padrão pré-existente e em outras ocasiões inventam novas abordagens para solucionar algo que já foi feito de outra forma.

A reutilização de conhecimento prévio de projetos bem sucedidos, criando-se modelos aplicáveis a novos problemas, é a base para alguns paradigmas que visam reaproveitar esforço passado em novas soluções. Esta abordagem abrange o uso integrado de metodologias, linguagens, programas, bibliotecas e componentes muitas vezes bastante heterogêneos para a obtenção de um resultado final que na maioria das vezes não poderia ser atingido não fosse desta forma. No entanto, a interconexão total de ferramentas disponíveis para o desenvolvimento de software ainda não é uma realidade, exigindo muitas vezes a utilização de diversas soluções independentes com um nível de acoplamento normalmente baixo.

O isolamento da complexidade e a reusabilidade permitem ocultar detalhes da implementação, reduzir a propensão a erros e facilitar os testes e a manutenção de um software. A criação de produtos com alto padrão de qualidade a um custo menor, demanda a produção em massa, ou pelo menos a produção múltipla de artefatos. Conclui-se que desta forma bons projetos podem ser facilmente modificados e reutilizados (BRAUDE, 2005).

Percebe-se hoje no mercado de desenvolvimento de software que ocorrem mudanças nos requisitos dos sistemas durante praticamente todas as fases do processo. Inclusive as metodologias ágeis de desenvolvimento de software apontam que arquitetos e programadores de sistemas precisam estar com a mentalidade aberta a acolher constantemente as mudanças de requisitos apresentadas pelos clientes. Segundo Beck (2004), XP (e*Xtreme Programming*) convida o cliente para ser uma parte integrante do time, com isto a especificação do sistema é continuamente refinada durante o desenvolvimento. O resultado é que os desenvolvedores precisam estar aptos à rapidamente responder às mudanças solicitadas.

Independente dos projetos seguirem metodologias ágeis ou metodologias mais tradicionais considera-se produtivo o uso de ferramentas de prototipação rápida e de base para o desenvolvimento de novos artefatos. Sobre a utilização de geradores de código para o uso em prototipação, McConnel (2005, p. 737) diz que

> Os geradores de código também são úteis para fazer protótipos de código de produção. Usando um gerador de código você pode montar em poucas horas um protótipo que demonstre os principais aspectos de uma interface com o usuário ou pode experimentar várias estratégias de projeto. Talvez demore semanas para codificar a mesma funcionalidade manualmente.

Para cada projeto de software, pode-se usar um conjunto de métodos e técnicas de acordo com a tecnologia utilizada e o problema a ser resolvido. Em alguns projetos parte do trabalho manual, maçante e sujeito a erros, pode ser implementado por softwares inteligentes. Em sua tese de doutorado, Czarnecki (1998) explica que a programação generativa centra-se na concepção e implementação de software reutilizável para geração de sistemas ao invés de escrevê-los do zero. Portanto o âmbito desta metodologia é a construção de famílias de sistemas.

Estudando diversas ferramentas de desenvolvimento de software encontram-se muitos recursos que visam reaproveitar um trabalho prévio para a construção de uma próxima etapa. Este reaproveitamento permite o aumento de produtividade facilitando tarefas repetitivas, seja fazendo parte do trabalho ou integrando-se com outras aplicações. Algumas

destas ferramentas permitem algum tipo de personalização para que os desenvolvedores possam aplicar de acordo com a necessidade de seus próprios projetos. Por outro lado, muitas outras não permitem tal nível de personalização. Disponibilizam apenas formas rudimentares de auxílio que acabam exigindo uma intervenção posterior com ajustes no código fonte que podem interferir na qualidade final do que foi produzido.

Também ocorre de determinados recursos mais completos de auxílio na escrita de código fonte estejam disponíveis apenas em um IDE (*Integrated Development Environment*) de uma linguagem específica. Isto facilita determinadas tarefas em algumas linguagens, mas não se torna aplicável em outras, mesmo que para tarefas similares.

Em trabalhos acadêmicos relacionados percebe-se que muitas propostas de ferramentas de geração de código fonte focam apenas para a obtenção de objetivos muito específicos. Em relação a ferramentas gratuitas ou comerciais também muitas delas tem uma única origem e um único destino, limitando a transformação de artefatos de software.

Sendo assim, a proposta deste trabalho é a construção de um *framework* com padrão aberto que possa ser altamente acoplável a novas especificações de entrada. Este conceito parte de um núcleo bem definido que usa conceitos de geração de código fonte, de forma que possa haver uma dinâmica entre definições de entradas e saídas. Busca-se a resolução de problemas genéricos de geração de código, através do reaproveitamento de algumas definições de entrada comuns pré-existentes, tais como uma especificação padrão XML. Pretende-se também que a ferramenta possa ser utilizada em vários projetos, independente de linguagem de programação, sistema operacional, metodologia de desenvolvimento ou ferramentas utilizadas, reduzindo a incompatibilidade entre ferramentas que acabam prejudicando a produtividade das equipes de desenvolvimento de software.

Este trabalho está dividido em cinco capítulos. No primeiro capítulo são apresentados os conceitos importantes para elaboração de um projeto de software reutilizável. No segundo capítulo é feito um estudo sobre geração de código fonte, sua estrutura, vantagens, cuidados e aplicações. No capítulo seguinte, são apresentados componentes especializados em geração de código e uma comparação de suas principais características, além da escolha de um dos componentes apresentados para ser utilizado na criação de um gerador de código. No quarto capítulo é apresentada a arquitetura proposta para a construção de um gerador de código fonte modular e expansível, bem como as ferramentas de desenvolvimento utilizadas. No último capítulo estão relacionados os casos de uso da ferramenta desenvolvida de acordo com a proposta deste trabalho.

1 PROJETO DE SOFTWARE

Segundo Braude (2005), ao projetar um software, é preciso enfrentar alguns desafios. Um dos maiores desafios é a tendência de mudança dos requisitos mesmo durante a construção da aplicação. Até mesmo alguns objetivos podem mudar a cada iteração. Na prática pode-se ter uma boa ideia da natureza de uma aplicação. No entanto, muitas vezes não sabe-se exatamente o que é desejado até que uma versão preliminar ou um protótipo seja produzido. Por esta razão, deve-se projetar permitindo que facilmente seja alterado aquilo que já foi produzido.

1.1 Objetivos do projeto de software

Para assegurar a qualidade, deve-se utilizar boas práticas de projeto e de programação. Frequentemente os objetivos de um projeto de software visam garantir a verificação da qualidade através de alguns padrões, que podem ser resumidos em alguns itens verificáveis:

- **Correção –** o primeiro objetivo de um projeto de software é satisfazer os requisitos da aplicação. A correção de um projeto de software consiste em atender aos requisitos, sendo que há abordagens formais e informais para garantir a correção. Uma das formas de avaliar é comparar o que foi desenvolvido está coerente com o que foi projetado em diagramas de caso de uso, atividades e classes. “Antes de tudo, somos responsáveis por assegurar que nossos códigos façam o que deve fazer” (BRAUDE, 2005, p. 53);

- **Robustez –** um projeto é robusto se for capaz de tratar condições adversas e interagir com o usuário de forma adequada em situações inesperadas, o software deve até mesmo a continuidade de forma seletiva em um processamento com os devidos tratamentos. Bons projetos devem prever suporte a condições anômalas;

- **Flexibilidade –** a adequação às mudanças, sejam elas acrescentar funcionalidades, modificar as existentes, ou eliminar algumas, devem ser implementadas sem que o contexto da aplicação mude. Padrões de projeto comportamentais visam garantir esta flexibilidade, permitindo encapsular comportamentos implementados de forma efetiva mas isolados. Para Gamma et al (2000), os padrões de projeto ajudam a evitar que a possibilidade de mudanças futuras esteja ligada a

necessidade de grandes reformulações. Desta forma deve-se projetar para que seja possível evoluir de maneira flexível;

- **Reusabilidade** – para Gamma et al (2000), projetar software orientado a objetos é difícil. Mas mais difícil ainda é projetar software reutilizável orientado a objetos. O autor também aponta que os projetos devem ser específicos para o problema a resolver, mas genéricos o suficiente para atender futuros problemas e requisitos. O processo de engenharia consiste na criação de produtos úteis com padrões de qualidade a custos mínimos muitas vezes produzidos em lote. Isto é uma exigência de mercado para diversos tipos de produto. No desenvolvimento de software não é diferente e muitas formas de reutilização são usadas, seja através de classes especializadas, componentes, *frameworks* ou API (*Application Programming Interface*);
- **Eficiência** – as funcionalidades de uma aplicação devem executar dentro de restrições de tempo conhecidas, ou seja, a otimização de tempo e a utilização de recursos devem ser equilibradas em relação aos demais critérios de qualidade. Para Braude (2005), um balanço mais complexo tem efeito ao se considerar projetos, explicando a facilidade relativa com que um projeto pode ser implementado e mantido. A Figura 1.1 mostra um balanço de três vias típico utilizado para definição de eficiência;

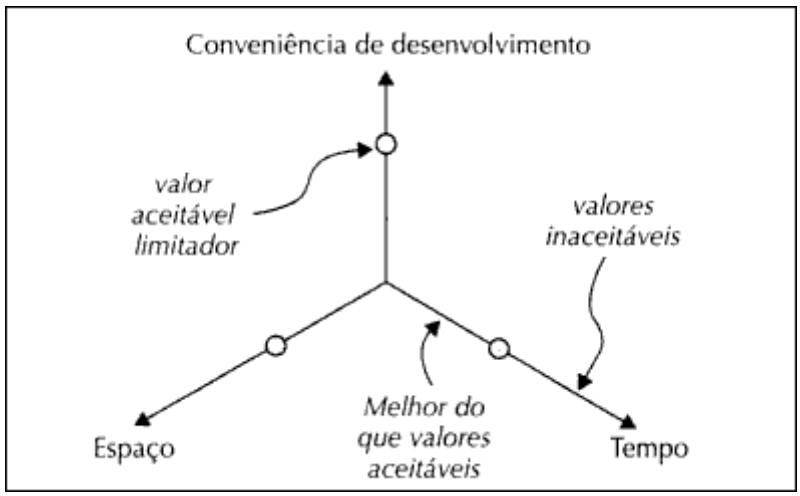

Figura 1.1 - Balanço entre espaço-tempo-desenvolvimento
Fonte: Braude (2005)

1.2 Padrões de projeto

Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto comum, a fim de torná-la útil para a criação de projetos reutilizáveis. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações, e a distribuição de responsabilidades (GAMMA et al 2000).

Não é realista esperar que uma aplicação esteja 100% livre de defeitos. No entanto, uma aplicação é confiável se estiver relativamente livre destes. A confiabilidade é uma questão que depende de inspeção e testes. O projeto afeta a confiabilidade pelo fato de que projetos transparentes tornam mais fácil produzir uma aplicação livre de erros. Já a usabilidade é considerada alta se os usuários acharem que a mesma aplicação é fácil de utilizar. (BRAUDE, 2005).

Gamma et al (1999 apud BRAUDE, 2005), classifica os padrões de projeto em três tipos:

- **Padrões de projeto Criacionais** - estes padrões de projeto ajudam a criar o projeto de aplicações que envolvem coleções de objetos: eles permitem a criação de várias coleções possíveis, e tratam propriedades tais como muitas versões de coleções e restrição dos objetos criados. Podem ser divididos em: *Abstract Factory, Builder, Factory Method, Prototype e Singleton*.
- **Padrões de projeto Estruturais** - ajudam a organizar conjuntos de objetos em formas com listas ou árvores vinculadas. Alguns padrões de projeto deste tipo podem ser: *Adapter, Bridge, Composite, Decorator, Façade, Flyweight e Proxy.*
- **Padrões de projeto Comportamentais** - cada padrão comportamental capta um tipo de comportamento em uma coleção de objetos e define como eles irão se relacionar mutuamente, ou seja, como será a comunicação entre estes objetos. Podem-se citar os seguintes padrões comportamentais: *Chain of Responsability, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor.*

1.3 Frameworks

O reuso é um dos objetivos mais importantes do desenvolvimento de software, sendo assim, produtos de alta qualidade são desenvolvidos reaproveitando partes de outras aplicações. Segundo Braude (2005), um f*ramework* é uma coleção de artefatos de software

que é utilizável por várias aplicações diferentes. Organizações que pensam em desenvolvimento futuro selecionam suas classes mais importantes e muito utilizadas como pertencentes aos seus *frameworks*. As aplicações podem usar os *frameworks* por meio de herança, agregação ou dependência e alternativamente podem comportar-se como aplicações genéricas que podem ser personalizadas.

A Figura 1.2 mostra o relacionamento entre classes do *framework*, classes de domínio da aplicação e classes de projeto restantes.

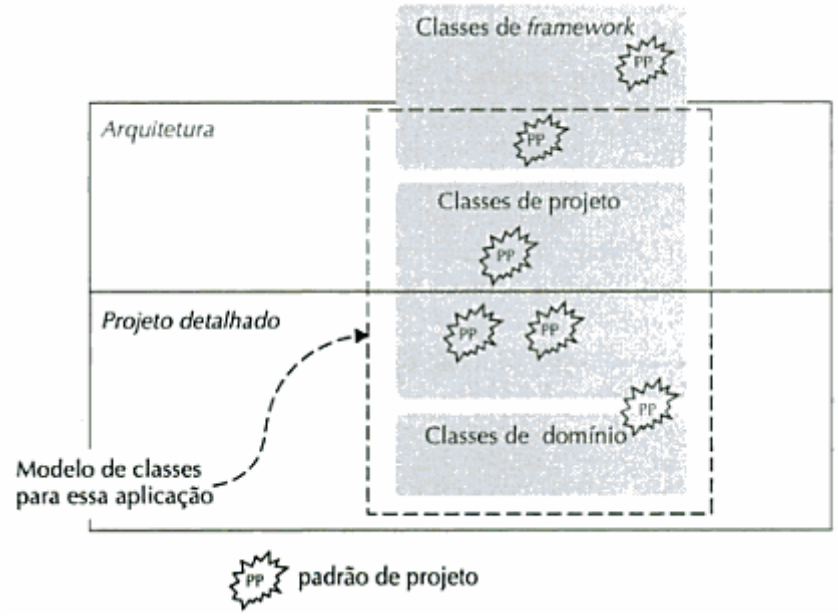

Figura 1.2 - Modelo de classes versus arquitetura e projeto detalhado
Fonte: Braude (2005)

Normalmente não se utiliza todas as classes de um *framework* em uma aplicação. As classes de projeto consistem em algumas exigidas pela arquitetura e outras que não são.

Para Gamma et al (2000, p. 42), "O *framework* dita a arquitetura de sua aplicação. Ele irá definir a estrutura geral, sua divisão em classes e objetos e em consequência as responsabilidades chave das classes de objetos, como estas colaboram e o fluxo de controle".

Para Braude (2005), os *frameworks* não precisam ser projetados somente se forem usados por um grande número de aplicações. É importante desenvolver uma camada de *framework* também uma aplicação única. Esse *framework* genérico define muitas outras classes que podem herdar da definição principal. A divisão em camadas genéricas promove melhores projetos. Resumindo, o objetivo dos *frameworks* é a reutilização de classes, relacionamento entre classes ou a definição de controle pré-programado.

1.4 Análise e desenvolvimento baseado em modelos

Existem várias metodologias de desenvolvimento de software que preveem que tanto o trabalho de análise, prototipação, implementação quanto a documentação sejam elaborados a partir de modelos. Frequentemente utiliza-se UML como uma notação padrão para diagramação. Ao desenhar um diagrama UML e ao programar de forma orientada a objetos, há influência direta na robustez, facilidade de manutenção e reusabilidade dos componentes de software (LARMAN, 2007).

Vários conceitos e arquiteturas acabam encontrando-se com a geração de código fonte em alguma etapa do desenvolvimento, mesmo que em abordagens distintas. Os conceitos citados a seguir serão explorados neste trabalho seja como referencial ou mesmo como metodologia voltada à aplicação de geração automatizada de código fonte.

1.4.1 MDA – *Model-Driven Architecture*

Segundo Mellor et al (2004), MDA, ou Arquitetura Dirigida a Modelos, remete a três conceitos básicos:

- Uma iniciativa da OMG (*Object Management Group*) para desenvolver padrões baseados na ideia de que modelagem é o melhor fundamento para desenvolver e manter sistemas;
- Uma marca para produtos que aderem a estes padrões;
- Um conjunto de tecnologias e técnicas associadas a estes padrões.

MDA prevê a descrição de um modelo com alto nível de abstração, conhecido como PIM (*Platform Independent Model*), na sequencia deriva modelos mais específicos chamados de PSM (*Platform Specific Model*) dos quais é possível extrair código fonte aplicável a projetos (OMG, 2011).

1.4.2 BPM – Business Process Modeling

Atualmente muito tem se falado em modelagem de processos de negócio como uma forma de enxergar como as tarefas de uma empresa se inter-relacionam entre si e com os sistemas de gestão. Facilitar o entendimento dos usuários bem como servir de fluxo de dados para o desenvolvimento de aplicações empresariais pode ser uma metodologia utilizada para a geração de protótipos ou mesmo partes funcionais de um software.

1.4.3 UML – Unified Modeling Language

Para Larman (2007), "A Linguagem de Modelagem Unificada, é uma linguagem visual para especificar, construir e documentar os artefatos dos sistemas". A palavra visual, nesta definição, é ponto chave, visto que UML é uma notação diagramática, padrão de fato, para desenhar ou apresentar figuras (com algum texto) relacionadas a um projeto de software. Por tratar-se de um padrão, UML é muito usada por diversas ferramentas de desenvolvimento de software para criar uma interatividade entre os envolvidos no processo.

Para Fowler (2003 apud LARMAN, 2007), há três modos de aplicar UML: o primeiro é o uso de UML como rascunho, onde os diagramas podem ser feitos mesmo à mão e visam representar um problema de difícil representação explorando o poder das linguagens visuais. O segundo modo é através do uso de UML como a planta de um software, ou seja, pelo detalhamento completo do projeto, podendo ser por engenharia reversa (sistema já existente) ou engenharia avante (geração de código). O último modo do uso de UML seria como uma linguagem de programação, onde o executável é gerado diretamente a partir dos modelos.

1.4.4 DSL – Domain-Specific Language

Segundo Fowler (2010, p. 27, tradução nossa), DSL, ou Linguagem de Domínio Específico, é "uma linguagem de programação de computadores de expressividade focada em um domínio particular". Fowler (2010), também indica que existem quatro elementos chave para esta definição:

- **Linguagem de programação** – uma DSL é usada para instruir computadores a executar algo;
- **Linguagem natural** – uma DSL deve ter um senso de fluência, onde a expressividade vem não apenas de expressões individuais, mas também da forma que elas podem ser compostas juntas;
- **Expressividade limitada** – uma linguagem de programação de propósito geral fornece muitas funcionalidades. Por outro lado, uma DSL suporta um mínimo de recursos necessários para apoiar o seu domínio. Não utiliza-se uma DSL para construir sistemas. No entanto pode-se usar uma DSL para um aspecto particular de um sistema;

- **Foco de domínio** – uma linguagem limitada só é útil se ela tem um foco claro em um pequeno domínio, este foco é que faz esta linguagem valer à pena.

Segundo Fowler (2010), algumas linguagens de *script*, expressões regulares, arquivos de configuração, e mesmo algumas definições escritas sob a sintaxe de um arquivo XML, podem ser considerados DSLs. Estas linguagens frequentemente são utilizadas em conjunto com outras tecnologias com a finalidade de representarem uma parte de um sistema, seja em modelagem, seja como linguagem interna de um *framework* ou API ou mesmo como auxiliar em algumas tarefas. Um exemplo de DSL pode ser o CSS que tem seu domínio específico na área de *web design,* que embora possa não ser escrito apenas por especialistas neste domínio, na maior parte das vezes são escritas por estes ou mesmo por ferramentas de *design* que geram o código CSS.

1.4.5 EMF – Eclipse Modeling Framework

O EMF é um *framework* de modelagem e facilitador de geração de código fonte para construção de ferramentas e outras aplicações baseadas e modelos de dados estruturados. Da especificação de modelos descritos em um XMI (*XML Metadata Interchange)*, o EMF provê ferramentas e suporte em tempo de execução para produzir um conjunto de classes Java a partir destes. Também possui classes que permitem a visualização e edição baseada em comando do modelo e um editor básico (ECLIPSE EMF, 2011).

1.5 Considerações Finais

Os conceitos estudados neste capítulo mostram a importância do uso de metodologias para elaborar um projeto de software que é concebido para ser reutilizável. Além da reutilização, a adição de novos recursos no futuro, mantendo uma estrutura consistente também é possível através da aplicação destes conceitos.

O estudo da estrutura dos *frameworks* é importante para este trabalho no sentido de entender a natureza expansível deste tipo de software. Se construído de forma estruturada levando em conta padrões de projeto e boas práticas de desenvolvimento, garante-se a qualidade e a facilidade de implementações de novos recursos que venham a agregar nas funcionalidades existentes. Uma ferramenta desenvolvida sob esta ótica poderá ter o seu uso expandido e até mesmo ser usado em outros projetos, visto que suas classes internas podem ser expostas de uma maneira reutilizável.

2 GERAÇÃO DE CÓDIGO FONTE

Em poucas palavras, geração de código fonte é um conceito sobre programas que escrevem programas. Com a complexidade das arquiteturas de desenvolvimento atuais, tais como Java *Enterprise Edition* (J2EE), Microsoft .NET e Microsoft *Foundation Classes* (MFC), cada vez mais é importante usar as habilidades das equipes de desenvolvimento de software para construir programas que auxiliem a construir aplicações. Genericamente, quanto mais complexo o *framework* de desenvolvimento, maior apelo para o uso de soluções de geração de código (HERRINGTON, 2003).

Muitos desenvolvedores perguntam-se porque tem que escrever tanto código para tarefas relativamente simples. Enquanto poderiam estar mantendo o foco no domínio de sua aplicação, muitas vezes passam horas construindo artefatos muito parecidos que são necessários para compor a aplicação final.

O uso de ferramentas que auxiliem a escrever partes da aplicação que resultam em código padronizado, livre de erros e com economia de tempo, significa muito ganho de produtividade (WALLS e RICHARDS, 2004).

"Um gerador é um programa que toma uma especificação de alto nível de uma peça de software e produz sua implementação. As peças de software podem ser um sistema de grande porte, um componente, uma classe, uma *procedure*, e assim por diante" (CZARNECKI e EISENECKER, 2000, p. 333, tradução nossa).

2.1 Estrutura básica

A maioria dos geradores de código encontrados na bibliografia e em produtos comerciais baseia-se em uma abordagem onde o desenvolvedor define uma linguagem de especificação, um analisador para esta linguagem e um transformador para converter a árvore sintática em artefatos. O diagrama da Figura 2.1 mostra esta abordagem.

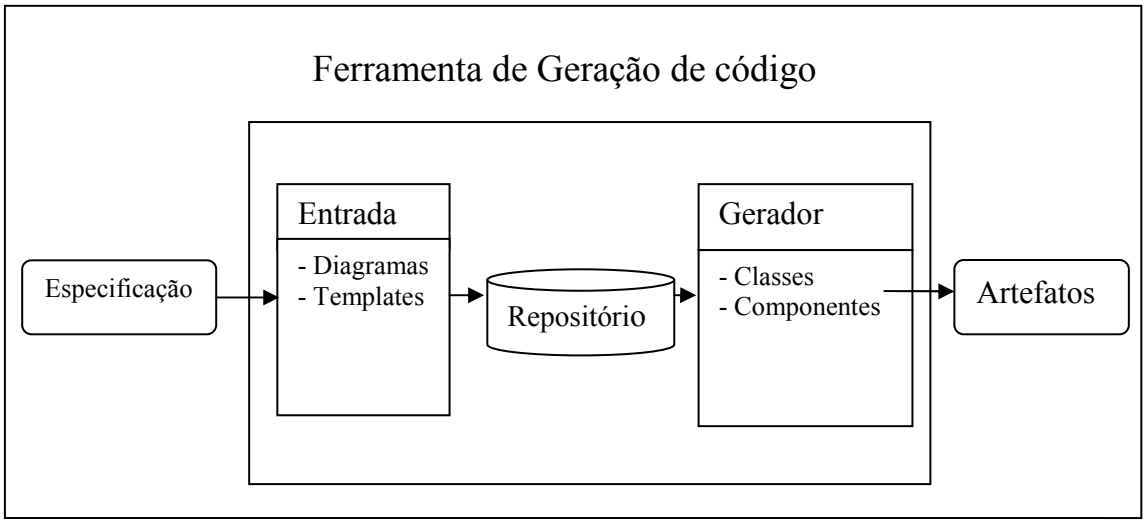

Figura 2.1 - Representação de uma ferramenta de geração de código
Fonte: Do próprio autor

Czarnecki e Eisenecker (2000), indicam que há três maneiras principais de construir geradores de código:

- Como programas isolados que conhecem toda a lógica e estruturação das entradas e das saídas. Este modelo seria o que reutilizaria menos outros recursos externos para chegar ao resultado final. Além disto não provê uma forma de expandir suas funcionalidades de uma forma flexível;
- Utilizando capacidade embutida de metaprogramação de algumas linguagens de programação. Nesse caso, como pode-se perceber, fica-se limitado a uma determinada tecnologia;
- Desenvolvendo geradores utilizando infraestrutura própria para este fim. Este seria o modo mais adequado de construir geradores, visto que permite um trabalho mais voltado para a transformação de código.

2.2 Vantagens

Para Herrington (2003), técnicas de geração de código garantem substanciais benefícios para a engenharia de software em todos os seus níveis. Estes benefícios são importantes não somente no âmbito técnico, mas são importantes também sob o ponto de vista gerencial, pois incrementam muito a produtividade e qualidade. A seguir, algumas das vantagens do uso destas técnicas:

- **Qualidade** – grande volume de código escrito à mão tende a ter qualidade inconsistente, visto que os desenvolvedores constantemente encontram novas abordagens para a escrita de código. A geração de código a partir de modelos cria uma consistência na base de código fonte instantaneamente. Quando os modelos são modificados e o gerador de código é executado novamente, correções de *bugs* e melhorias no código fonte são aplicadas consistentemente;
- **Consistência** – código fonte construído por um gerador de código é consistente com o projeto de API e utilização de nomenclatura de variáveis. O resultado é uma interface sem surpresas que é fácil de entender e utilizar;
- **Centralização do conhecimento** – a modificação na modelagem de um sistema durante sua produção, mesmo nos melhores sistemas codificados manualmente pode ter um grande impacto. A simples troca da nomenclatura de uma tabela de um banco de dados envolve modificações manuais no esquema físico, na camada de acesso a dados, na documentação e nos testes unitários. Uma arquitetura voltada à geração de código fonte permite que a troca de nome da tabela dada como exemplo, ocorra em um único local e a partir de nova geração do código fonte todos os pontos passam a tratar pela nomenclatura atualizada;
- **Mais tempo de projeto** – cronogramas para projetos baseados em geração de código fonte podem ser significantemente diferentes de projetos codificados de forma totalmente manual. Nos projetos baseados em codificação manual que tenham muito volume de código, muito cuidado é necessário para analisar a melhor forma de uso de APIs, visto que algumas mudanças podem ter impacto substancial com reescrita em muitos trechos de código. Com a abordagem de geração de código, os desenvolvedores podem reescrever os modelos modificando a forma como as APIs são usadas constantemente, visto que grandes trechos de código poderão ser escritos novamente em pouco tempo, permitindo facilmente testar o desempenho de novas abordagens no sistema como um todo. Devido a esta facilidade é possível investir mais tempo em modificações no projeto da arquitetura e definição de protótipos, evitando-se desperdício de tempo e retrabalho;
- **Independência de tecnologia** – arquiteturas de geradores de código permitem que especificações sejam criadas em alto nível, em definições independentes de

linguagem de programação. A abstração da semântica de aplicação do código fonte de destino garante que novos modelos sejam criados para traduzir as lógicas de negócio, definições de dados e de interface com o usuário facilmente para outras plataformas de forma muito mais fácil do que se tivessem sido escritas manualmente;

- **Qualidade da documentação** – a documentação do código fonte gerado automaticamente pode ser muito mais consistente e detalhada do que quando escrita manualmente. Caso os modelos sejam bem documentados é possível incluir no código fonte, por exemplo, trechos de documentação no estilo Javadoc, garantindo alta qualidade de documentação e entendimento da arquitetura do sistema;
- **Motivação das equipes** – projetos longos podem ser difíceis para equipes de desenvolvimento, e projetos longos com grande volume de codificação tediosa pode ser muito pior. A geração de código reduz tempo das etapas de cronograma onde grande volume de codificação repetitiva seja necessário, permitindo aos profissionais manterem-se focados e interessados nas regras de negócio, além de ter orgulho da qualidade, uniformidade e confiabilidade nos trechos de código gerados automaticamente;
- **Agilidade no desenvolvimento** – um conceito chave da geração de código é a maleabilidade, que significa que o software pode ser facilmente modificado e incrementado durante o ciclo de desenvolvimento. Com esta garantia, é possível incentivar ciclos de produção e entrega para o usuário final sem perda de produtividade.

2.3 Cuidados

Assim como outras metodologias de desenvolvimento não abrangem a integralidade do processo de construir software com qualidade, a geração de código fonte também não, e precisa ser usada com alguns cuidados e limites. Walls e Richards (2004), chamam estes cuidados de sabedoria da geração de código, e fornecem algumas dicas importantes para integrar de forma correta o código gerado no processo de desenvolvimento:

- **Não gerar o que não se conhece** – a geração de código é uma ferramenta para bons desenvolvedores tornarem-se melhores. Antes de considerar gerar

determinadas peças de software, deve-se garantir que há um bom conhecimento da tecnologia que está sendo utilizada, inclusive tendo consciência de cada instrução que será gerada, do desempenho que terá e todos os reflexos no restante da aplicação. Não se deve gerar algo que quando ocorrer um problema os desenvolvedores descubram que não sabem o que aquilo realmente faz;

- **Testar o código gerado** - mesmo em organizações com um forte compromisso com testes, muitas vezes há relutância em escrever testes de unidade para o código gerado. Talvez esta resistência tenha origem na confiança no gerador de código. No entanto, geradores de código também são programas e não são perfeitos. Logo deve-se por à prova tudo o que for gerado, permitindo que haja a certeza que o código gerado faz aquilo que pretende fazer e que continuará a fazê-lo da mesma maneira no futuro;

- **Mudar o projeto para comportar código gerado** – muitas vezes ao analisar um determinado sistema, pode-se descobrir que não é possível empregar a geração de código. A tendência em muitos projetos pode ser a de desistir da geração de código por ignorar o fato de que, com algumas pequenas alterações no projeto do sistema passa a ser possível a utilização desta técnica. Algumas vezes as mudanças são pequenas, talvez nomes de classe ou métodos precisem ser alterados para fins de consistência, ou talvez seja preciso mudar a hierarquia de objetos ligeiramente para acomodar as necessidades do gerador. Na maioria das vezes o benefício de ser capaz de gerar o código supera em muito o custo de fazer tais alterações;

- **Aplicar a geração de código por camada -** quando decide-se por aplicar a geração de código para um sistema existente, não recomenda-se tentar converter todas as camadas da aplicação de uma só vez. Em uma aplicação J2EE, por exemplo, pode-se começar com a camada EJB. Uma vez que o código gerado esteja bastante estável, então pode-se considerar a geração da camada Web. Isolar as camadas dá mais controle e mantém o processo de migração mais gerenciável com um resultado mais confiável;

- **Manter separado o código gerado –** código gerado pode em alguns casos ser considerado apenas um subproduto de outros processos. Nestes casos, eles devem ser colocados em repositórios separados do código escrito manualmente, em

determinadas aplicações o código gerado poderia até mesmo ser eliminado após a compilação, sendo gerado novamente a cada vez que o sistema seja reconstruído, garantindo a consistência entre a modelagem que o deu origem e seu resultado final. Mesmo que essa abordagem não seja utilizada, uma boa prática é manter no mínimo uma separação em pacotes ou diretórios distintos;

- **Evitar redundância de código –** um ponto importante é que deve-se tentar evitar a redundância de código. Aquilo que parece repetitivo, que aparece com frequência em muitas peças de código gerado, deve ser isolado. Sempre que possível deve-se usar recursos de orientação à objetos tais como a herança para que o código gerado implemente apenas o que é específico.

2.4 Aplicações

Diferentes tipos de geração de código são usados para resolver problemas distintos, Herrington (2003), explica que existem várias técnicas que visam atender a situações específicas, elas são divididas de acordo o tipo de camada ou parte de um projeto de software em que são aplicadas, entre elas pode-se citar as seguintes:

- **Geração de interface gráfica** – a geração de interface gráfica com usuário, também conhecida como UI (*User Interface*) ou GUI (*Graphical User Interface*), tais como formulários Web ou telas para o uso em aplicações *desktop*, são passíveis de geração automática. Herrington (2003) comenta que alguns engenheiros de software tendem a afastarem-se destas soluções porque pensam que o resultado não será tão bom, ou mesmo imaginam um resultado inutilizável. No entanto com alguns cuidados é possível beneficiar-se desta técnica obtendo uma interface de usuário bastante consistente e com uma usabilidade bastante padronizada. Outra grande vantagem da geração de interface gráfica é que de certa forma obriga-se que a arquitetura do sistema seja projetada para a separação das regras de negócio e da camada de apresentação, o que é uma boa prática na construção de qualquer aplicação;

- **Geração de documentação** – muitas linguagens de programação prometem um código autodocumentado, ou seja, um código tão claro que os desenvolvedores podem entender o seu propósito apenas o lendo. Na prática, um código com documentação tem seu entendimento facilitado, e se bem elaborado, é uma importante ferramenta de produtividade. McConnel (2005), dedica um capítulo

inteiro de seu livro sobre boas práticas no desenvolvimento de software para a documentação de código, e neste é enfático da necessidade de manter comentários nas declarações de estruturas de dados e comportamento dos programas, sejam variáveis, funções, métodos entre outros. Também indica que deve-se tirar proveito de utilitários de documentação de código, tal como o Javadoc. O código fonte gerado automaticamente também deve conter documentação. No entanto é preciso que a fonte destas informações esteja presente na origem para que seja inserida no resultado final. Também é possível extrair somente os dados de documentação de uma determinada origem, isto seria útil em alguns projetos onde não é possível utilizar um utilitário de documentação de código. Por exemplo, é possível através de técnicas de geração de código extrair uma documentação em HTML a partir de comentários padronizados em estruturas de dados SQL;

- **Geração de testes unitários** – o comportamento de execução de funções de um software pode ser verificado a partir da execução de programas escritos especificamente para este fim. A execução deve buscar por defeitos no código, que apresentem resultados diferentes do esperado. Testes unitários são importantes para garantir a qualidade final. Sobre testes unitários, Pressman (2010, p. 295) comenta que

> "O teste de unidade focaliza o esforço de verificação na menor unidade de projeto de software – o componente ou módulo de software. Usando a descrição de projeto no nível de componente como guia, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo."

Existem várias APIs próprias para a elaboração destes testes que podem ser aplicadas a diversas linguagens de programação: JUnit para Java e DUnit para Delphi podem ser usadas como alguns exemplos bastante populares. Tais APIs permitem que as funções sejam testadas e o resultado de execução seja colocado à prova sob um resultado esperado, gerando relatórios de execução ao final, apontando o sucesso ou falha do conjunto inteiro. Algumas IDEs oferecem recursos para a geração do código base para a elaboração de testes unitários. Mas muitas linguagens de programação e ambientes de desenvolvimento não oferecem nenhum tipo de auxílio para isto, nestes casos é possível usar geração de código para gerar programas que executem as funções de um software, ficando a cargo do desenvolvedor apenas a inclusão de diversas chamadas com seus respectivos resultados esperados;

- **Geração de programas para acesso a arquivos** – embora seja possível em qualquer linguagem construir funções ou usar bibliotecas prontas para acessar alguns tipos de arquivos com formato pré-definido, com técnicas de geração de código, é possível eliminar a necessidade de construir uma biblioteca complexa para tratamento do formato dos dados, além de permitir uma fácil portabilidade para qualquer linguagem de programação, apenas mudando a sintaxe de escrita do código que faz a manipulação de I/O;

- **Geração de manipuladores de dados** – código SQL escrito diretamente dentro de um programa que trate diretamente a regra de negócio ou da interface com o usuário não é uma boa prática de programação, a propagação de código escrito em vários programas é mais contra indicado ainda. Um dos maiores problemas é que pelo fato do código SQL não ser compilado juntamente ao programa, os resultados de um erro de sintaxe podem aparecer somente no momento que o mesmo for executado, e se este código for montado dinamicamente o problema é ainda maior. O ideal é que exista uma camada responsável pela persistência e consulta de dados, onde todo o código de acesso a dados esteja centralizado. A informação correta sobre as estruturas de dados de um sistema podem ser extraídas de metadados dos próprios bancos de dados, de modelos conceituais ou de dicionários de dados. Com estas informações, é possível gerar código para manipular dados, criando-se funções para operações básicas de inserção, alteração e deleção, bem como busca por dados a partir dos índices principais e também criar funções para a elaboração de consultas específicas que sejam elaboradas de forma otimizada. De acordo com Kreisig (2007), parte dos objetos do framework jFace da empresa Quatro Informática Ltda, que são responsáveis pela parte de persistencia dos dados são criados automaticamente por um gerador de artefatos, que efetua a geração a partir de um modelo E-R (Entidade-Relacionamento).

- **Geração de *Web Services Layer* –** um *Web Service* expõe a lógica de negócios de uma aplicação para o mundo externo. Se as camadas de lógica de negócio e de segurança forem bem definidas, a tarefa de exportá-las através de um *Web Service* requer um código para criar uma interface entre a API e a camada RPC e organizar o tráfego de dados entre as duas. Este processo envolve a criação de uma camada de acesso remoto que permite a ligação com o usuário final. Isto pode ser feito em uma grande variedade de sistemas operacionais e linguagens de

programação. Então independente da tecnologia que o *Web Service* tenha sido criado, é possível efetuar a geração de código das definições de interface com tais serviços, seja a aplicação cliente escrita em Visual Basic, Perl, Java, C#, entre outras;

- **Geração de *header files*** – a geração de arquivos *header* em C++ é uma tarefa simples, porém repetitiva e sujeita a inconsistências. É possível usar uma técnica de geração de código que analise o código fonte e mantenha atualizados os arquivos .HPP de forma automatizada com as assinaturas dos métodos da aplicação;

- **Criação de *Wrappers* para DLLs** – um método clássico de escrita da código reutilizável para Windows é a criação de DLLs (*Dynamic-Link Library*), que são bibliotecas de ligação dinâmica, ou seja, conjuntos de funções que podem ser escritos em uma determinada linguagem de programação e usados por muitas outras. Um dos problemas com DLLs é que a sua interface é funcional, logo não possui uma interface orientada a objetos. Isto requer que a interface de objetos seja colocada sob um ponto de vista procedural. Para cada método no objeto que deseja-se exportar precisa haver uma função na interface DLL. Com esta técnica é possível gerar em qualquer linguagem as estruturas de dados e de chamada e retorno de cada função da DLL, empacotando esta interface em um programa ou classe específica da tecnologia que fará uso da mesma;

- **Criação de funções *lookup*** – em algumas situações, para ganhar desempenho em algumas aplicações, é preciso criar funções *lookup*. Estas funções nada mais são que um repositório de alguns resultados conhecidos para um determinado cálculo que armazenados prontos tem desempenho bem maior do que se for calculado a cada chamada. A ferramenta de geração de código precisa apenas implementar o cálculo e colocar os resultados em uma estrutura de dados que permita o retorno instantâneo do mesmo quando a função for executada;

- **Criação de funções de regra de negócios** – mesmo as regras de negócio representando a parte mais dinâmica e customizada de escrita de uma aplicação, são passíveis de terem seu código gerado automaticamente. Talvez nem todas as regras de negócio possam ser geradas. Mas com a utilização de boas ferramentas de modelagem de processos de negócio, é possível modelar cálculos, decisões,

desvios condicionais, entre outros elementos que definem o comportamento das funções do programa a ser gerado.

2.5 Modificação do código gerado

Sobre a modificação do código fonte gerado, Fowler (2010) explica que existem dois tipos de geração de código. O primeiro tipo é gerado como um "primeiro-passo", e é aquele em que o código fonte resultante é uma base para implementações a partir de um modelo, é gerado uma única vez e pode ser modificado manualmente. O segundo tipo é aquele em que o código gerado nunca deve ser modificado diretamente pelos desenvolvedores, exceto por adições relacionadas a testes ou depuração.

Um exemplo bastante simples de geração de código fonte como base para uma implementação maior é o recurso existente na IDE Eclipse, conhecido nesta ferramenta simplesmente por *templates* (modelos), onde a partir da digitação de uma determinada palavra, seguida de uma combinação de teclas, o editor expande o código fonte gerando trechos maiores onde o programador posteriormente insere o restante do código. Por exemplo, digitando-se:

```
sysout <CTRL>+<ESPAÇO>
```

É gerado o seguinte trecho de código:

```
System.out.println();
```

Ou

```
switch <CTRL>+<ESPAÇO>
```

É gerado o seguinte trecho de código:

```
switch (key) {
case value:

     break;

default:
     break;
}
```

Outro exemplo desta forma de geração de código fonte pode ser visto em ferramentas de modelagem como o ArgoUML. Na ferramenta citada, a partir da elaboração de um diagrama de classes é possível gerar o código que será usado como base para a implementação das classes. Devido aos poucos recursos oferecidos para customização é preciso que o código resultante seja modificado manualmente. Um dos motivos é que quando

se utiliza agregação, os objetos são criados sempre herdando da classe Vector, caso o desenvolvedor decida implementar com outro tipo de coleção, precisará necessariamente modificar o código fonte. Walls e Richards (2004), definem este estilo de geração de código como "*passive code generation*", ou seja, geração de código passiva. A Figura 2.2 representa essa forma de geração de código, onde o código é gerado uma única vez e posteriormente é alterado pelo desenvolvedor.

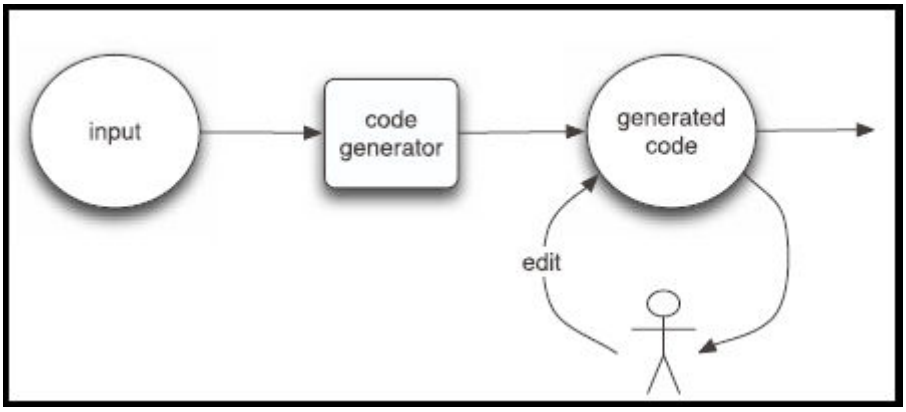

Figura 2.2 - Geração de código passivo
Fonte: Adaptado pelo autor, segundo Walls e Richards, 2004

Já para o estilo de geração de código fonte que após gerado não pode ser modificado pelo desenvolvedor, para Walls e Richards (2004), é considerado como "*active code generation*", ou seja, geração de código ativo. A modificação neste código só será possível com outras iterações de geração posteriores. A Figura 2.3 mostra o fluxo das ações com este estilo de código gerado.

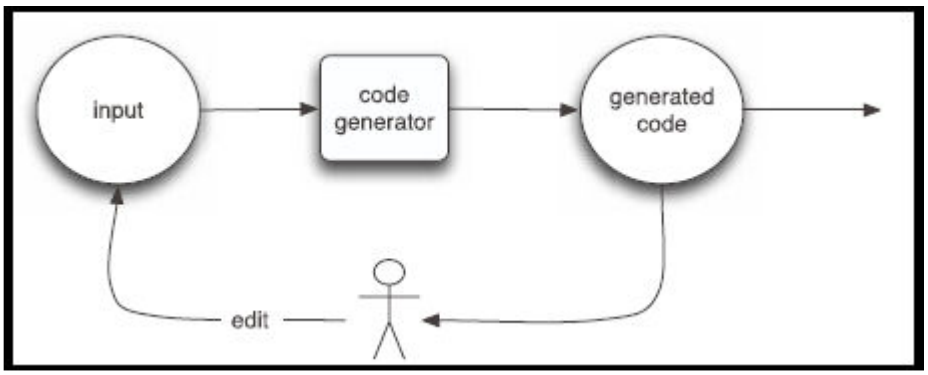

Figura 2.3 – Geração de código ativo
Fonte: Adaptado pelo autor, segundo Walls e Richards, 2004

Um dos exemplos de geração de código ativo é usado na IDE Netbeans, onde ao ser elaborada uma interface com usuário, por exemplo um JFrame com alguns componentes, o trecho de código gerado pela IDE é inserido no código fonte da aplicação com o seguinte

comentário: “// <editor-fold defaultstate="collapsed" desc="Generated Code">”. Este comentário funciona como marcação para o editor esconder o trecho de código gerado e evitar alterações indevidas, visto que qualquer nova modificação em tempo de projeto irá gerar novamente o código, sobrescrevendo qualquer modificação efetuada manualmente.

A Figura 2.4 mostra trecho de código gerado pela IDE Netbeans com um comentário avisando para que o código não seja modificado, pois o conteúdo do método é sempre regerado pelo editor de formulários.

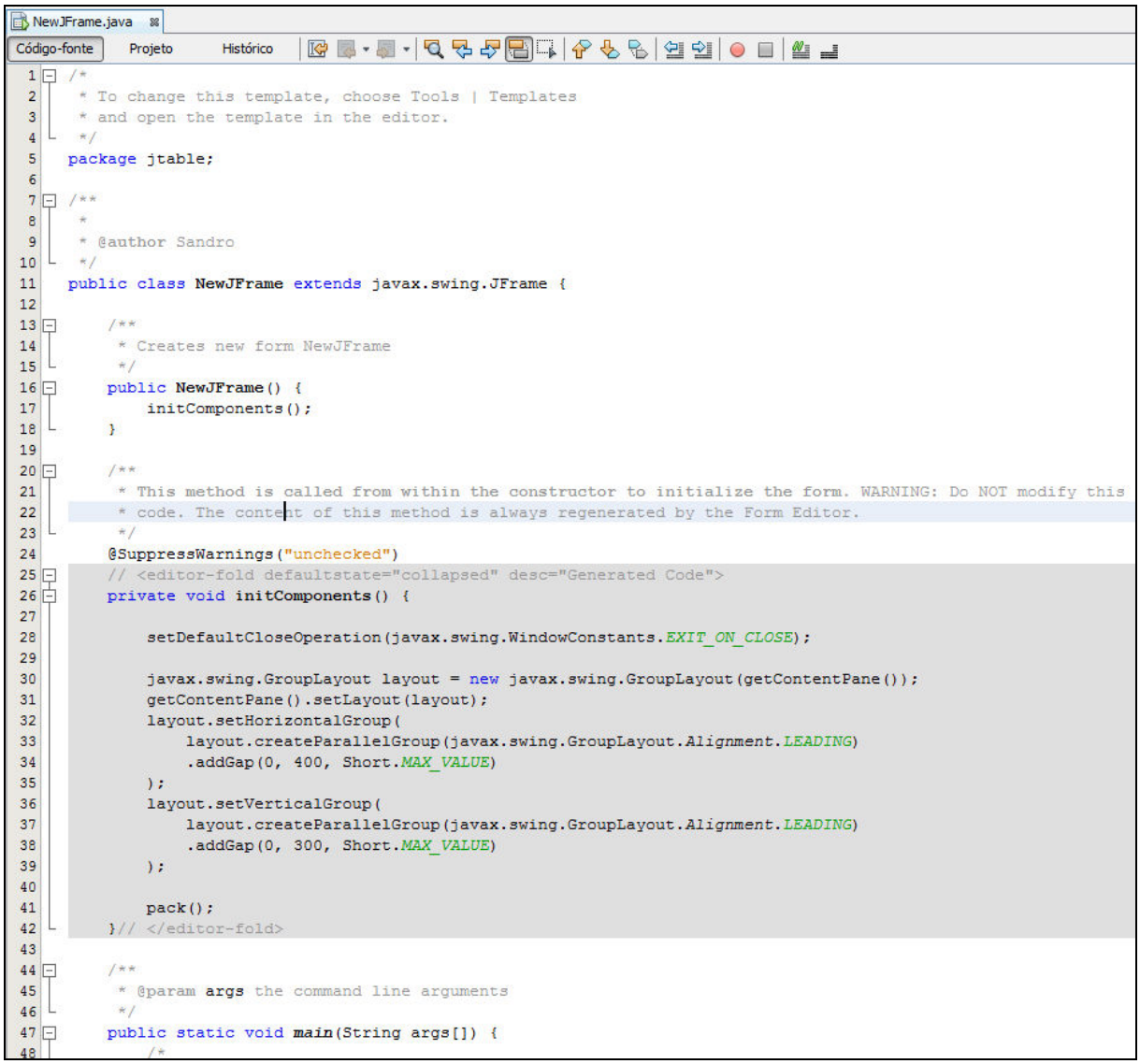

Figura 2.4 – Código gerado pelo Netbeans
Fonte: Do próprio autor

2.6 Ferramentas de geração de código existentes

Existem algumas ferramentas similares à que é proposta neste trabalho. Estas ferramentas trabalham com geração de código fonte genérico, a vantagem com seu uso é não precisar desenvolver uma ferramenta específica. No entanto apresentam algumas limitações, tais quais dependência de sistema operacional, escopo limitado, não permitir expansão customizada, alto custo, código fechado, curva de aprendizado elevada e pouca disseminação. Algumas das ferramentas que podem ser citadas são:

- **ArgoUML** – gera classes de objetos modelados em Java em C++. No entanto não permite a customização das saídas geradas, exceto pela utilização de um *plugin* de impressão de terceiros que efetua geração de código a partir de *templates engines;*
- **FireStorm DAO –** esta ferramenta é um gerador de código Java que auxilia os desenvolvedores de software a produzirem automaticamente as rotinas DAO (*Data Access Object*) para acesso à bancos de dados relacionais. A partir de engenharia reversa com conexão JDBC (*Java Database Conectivity*) lê os objetos de um banco de dados e permite a criação de programas que fazem o acesso destes dados. Existe uma versão paga sem opção para download para avaliação no *site* do desenvolvedor que indica que possui suporte a customização, mesmo assim o escopo da ferramenta é limitado;
- **CodeSmith Generator** – é uma ferramenta de geração de código que utiliza customização de *templates*, com sintaxe praticamente idêntica a ASP.NET, possui boa interface gráfica com colorização de sintaxe de *template* e de código gerado, que pode ser em C#, Java, VB, PHP, ASP.NET, SQL, etc. CodeSmith é uma ferramenta proprietária e escrita apenas para ser executada na plataforma Windows, integra-se via *plugins* com o Visual Studio.

3 TEMPLATES ENGINES

Templates são modelos de código, baseados em uma estrutura de um programa já conhecido e experimentado. Geralmente os *templates* são arquivos que possuem partes fixas que serão geradas no código fonte tais quais estão escritas. Os *templates* também contemplam *tags* (marcadores), que são usadas para serem substituídas pelo conteúdo dinâmico oriundo das especificações de entrada. Para Herrington (2003), o uso de *templates* permite que a formatação do código fique separada da lógica que determina o que deve ser gerado. Esta separação entre a definição da lógica do código e sua formatação é uma boa forma de abstração.

A Figura 3.1 mostra o fluxo de entrada e saída de um gerador simples que utiliza *templates*. O gerador lê o arquivo de definições das especificações de entrada e a mescla com a formatação do *template* para a obtenção do código fonte gerado.

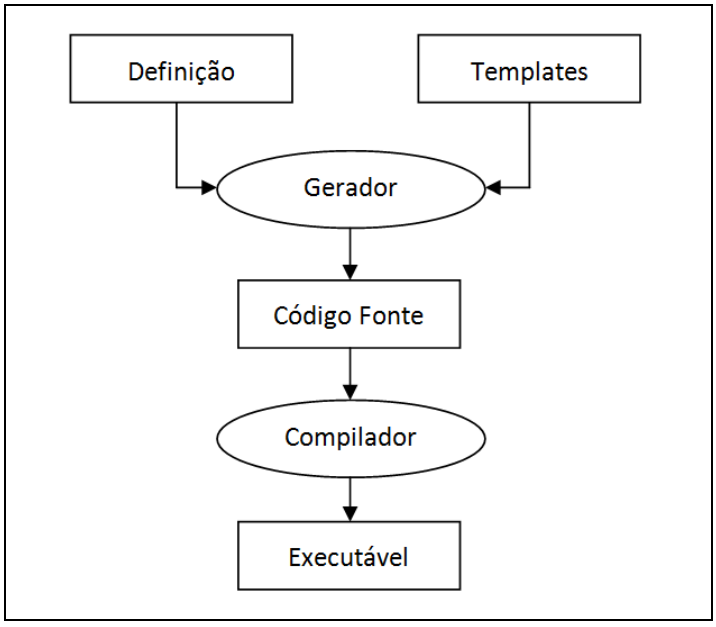

Figura 3.1 – Fluxo de entrada e saída de um gerador com *template*
Fonte: Herrington, 2003, traduzido pelo autor

Um *template engine*, ou motor de *template*, é um componente especializado em reconhecer uma estrutura de *template* e mesclar uma especificação de entrada ao mesmo, obtendo uma saída. Em aplicações Web, existem muitas tecnologias que implementam este conceito, por exemplo, ASP.net, PHP, entre outras. Para a finalidade de geração de código independente de tecnologia ou plataforma de destino, existem alguns componentes bem específicos que são leves, portáveis e poderosos. Estes componentes são utilizados em conjunto com muitas ferramentas bastante populares, como as IDEs Eclipse e Netbeans. O

uso de um *template engine* é bastante facilitado por estar abstraído geralmente em uma API bem documentada. Também são bem difundidos, encontram-se facilmente exemplos em *sites* Web, artigos e uma boa oferta de livros sobre o assunto. O uso destes componentes facilita o desenvolvimento de aplicações para geração de código fonte e incrementa a qualidade do produto final. As próximas seções apresentam os *template engines* que foram estudados neste trabalho: FreeMarker, Velocity e XDoclet.

3.1 FreeMarker

FreeMarker é um *template engine*, ou seja, uma ferramenta genérica para gerar saídas de texto formatadas: qualquer coisa desde HTML até qualquer tipo de código fonte gerado. O FreeMarker é um software baseado em manipulação de *templates*. Ele é apresentado sob o formato de um pacote Java, que é utilizado como uma biblioteca de classes para ser utilizada por outros programas Java. FreeMarker não é uma aplicação para usuários finais, mas algo que engenheiros de software possam embutir em seus produtos (FREEMARKER, 2011).

FreeMarker foi projetado para ser prático para a geração de páginas Web HTML, particularmente por aplicações baseadas em *Servlets* seguindo o modelo MVC. A ideia por trás do uso do padrão MVC para criar páginas dinâmicas para a internet, é a separação do *design* de HTML da lógica de negócios escrita por programadores. Cada um trabalha naquilo em que é melhor. Web *designers* podem trocar a aparência da página sem que os programadores tenham que fazer qualquer mudança no código fonte ou recompilar nenhum trecho de código. Isto é possível porque a lógica da aplicação escrita em programas Java e o *design* das páginas que ficam nos *templates* FreeMarker são separados. A vantagem do uso de *templates* neste caso, é que eles não são poluídos com complexos fragmentos de programas. Esta separação é útil mesmo em pequenos projetos onde programador e o autor das páginas HTML são a mesma pessoa, pois esta abordagem ajuda a manter a aplicação clara e com fácil manutenibilidade.

Embora o FreeMarker tenha alguma capacidade de programação, ele não é uma linguagem de programação plena como PHP, C ou Delphi. A lógica de programação é implementada em programas Java que preparam os dados que serão formatados e exibidos (por exemplo, consultas SQL), e o FreeMarker apenas gera saídas textuais que mostram o conteúdo da páginas que foram preparados usando os dados em conjunto com os *templates*. Netbeans, Alfresco, ZeroCode, FMPP, Mango e DBSight são algumas aplicações que utilizam

o FreeMarker como componente de geração de código ou para criação da camada de apresentação.

A Figura 3.2 mostra um exemplo de criação de páginas HTML com conteúdo dinâmico, gerado a partir de objetos Java e *templates*. Nesta figura é possível verificar o funcionamento básico do FreeMarker.

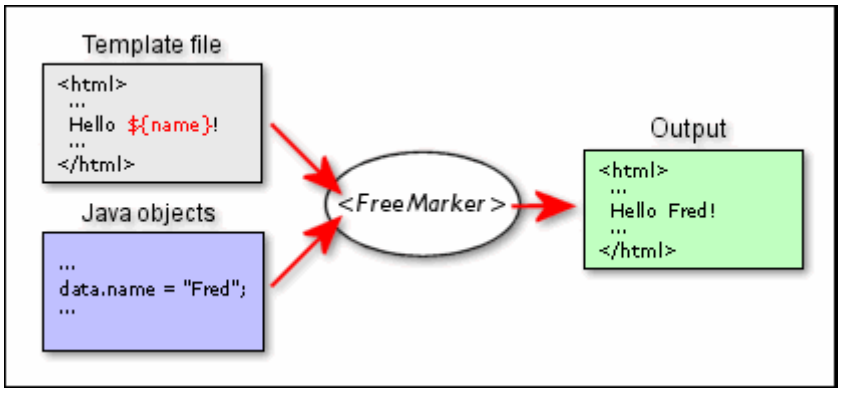

Figura 3.2 – Criação de páginas HTML com FreeMarker
Fonte: FreeMarker (2011)

3.1.1 FreeMarker Template Language

O FreeMarker possui uma linguagem para escrever os *templates,* que implementa elementos denominados "diretivas". Com estas diretivas é possível elaborar uma estrutura lógica adequada a mesclar dados e *templates*, gerando código fonte. O Quadro 3.1 - Diretivas da linguagem de *template* do FreeMarker apresenta as principais diretivas usadas pela linguagem de *template* do FreeMarker.

Diretiva	Sintaxe	Propósito
#if #else #elseif	<#if *condition*> ... <#elseif *condition2*> ... <#elseif *condition3*> ... ... <#else> ... </#if>	Diretivas utilizadas para testes condicionais. Testes aninhados também são válidos.
#switch, #case, #default, #break	<#switch *value*> <#case *refValue1*> ... <#break> <#case *refValue2*> ... <#break> ... <#case *refValueN*>	Switch é usado para escolher um fragmento de código que deve ser gerado dependendo do valor de uma expressão.

Diretiva	Sintaxe	Propósito
	... <#break> <#default> ... </#switch>	
#list, #break	<#list *sequence* as *item*> ... </#list>	Pode ser usado para processar a sessão de um *template* para cada variável contida em uma sequência ou uma *collection*. Para cada iteração do *loop* a variável de *loop* (item) contém o valor corrente de uma ocorrência da sequência.
#include	<#include *path*> or <#include *path options*>	Utilizado para inserir outro *template* FreeMarker no *template* atual em tempo de execução.
#import	<#import *path* as *hash*>	Importa uma biblioteca de comandos, ou seja, executa o *template*, mas sem acrescentar nada na saída. É usado para avaliar variáveis, popular estruturas entre outras ações.
#noparse	<#noparse> ... </#noparse>	Um trecho de código incluído dentro desta diretiva não é interpretado e executado pelo FreeMarker, ou seja, seu conteúdo é copiado para a saída.
#compress	<#compress> ... </#compress>	Remove espaços em branco duplicados.
#escape, #noescape	<#escape *identifier* as *expression*> ... <#noescape>...</#noescape> ... </#escape>	Quando um trecho do *template* é marcado com esta diretiva, as diretivas de interpolação (${...}), são combinadas com *expression*.
#assign	<#assign *name=value*> or <#assign *name1=value1*	Atribui ou altera o valor de uma variável ou estrutura de dados.

Diretiva	Sintaxe	Propósito
	name2=value2 ... nameN=valueN> or <#assign *same as above...* in *namespacehash*> or <#assign *name*> *capture this* </#assign> or <#assign *name* in *namespacehash*> *capture this* </#assign>	
#global	<#global *name=value*> or <#global *name1=value1 name2=value2 ... nameN=valueN*> or <#global *name*> *capture this* </#global>	Similar a diretiva *assign*, com a diferença que a visibilidade da variável será global, e não somente em um *namespace*.
#local	<#local *name=value*> or <#local *name1=value1 name2=value2 ... nameN=valueN*> or <#local *name*> *capture this* </#local>	Similar a diretiva *assign*, com a diferença que a visibilidade da variável será local, somente dentro de uma definição de macro ou de uma função.
#setting	<#setting *name=value*>	Modifica uma configuração do FreeMarker a partir daquele ponto.
User-defined directive (<@...>)	<@*user_def_dir_exp ...*> ... </@*user_def_dir_exp*> or <@*user_def_dir_exp ...*> ... </@>	Executa uma diretiva de usuário: uma macro ou uma função.
#macro, #nested, #return	<#macro *name param1 param2 ... paramN*> ... <#nested *loopvar1, loopvar2, ..., loopvarN*> ... <#return> ... </#macro>	Cria uma *macro*, que armazena um fragmento do *template* que pode ser usado como uma diretiva de usuário.
#function, #return	<#function *name param1 param2 ... paramN*> ... <#return *returnValue*> ... </#function>	Cria uma função que retorna um determinado valor.

Diretiva	Sintaxe	Propósito
#flush	<#flush>	Garante que o conteúdo de saída da execução que está em *buffer* seja enviado para disco.
#stop	<#stop> or <#stop *reason*>	Aborta o processamento, utilizado para situações de exceção.
#ftl	<#ftl *param1=value1 param2=value2 ... paramN=valueN*>	Diretiva de informação que o arquivo trata-se de um *template* FreeMarker, também permite configurar os parâmetros de execução.
#t, #lt, #rt	<#t> <#lt> <#rt>	Diretivas para ignorar espaços em branco na linha.
#nt	<#nt>	Significa "*no-trim*", ou seja, desabilita o efeito de outras diretivas *trim*.

#attempt, #recover	<#attempt>
	  attempt block
	<#recover>
	  recover block
	</#attempt>
		Tratamento de exceções, caso ocorra algum erro dentro de "*attempt block*", então nada do conteúdo deste bloco é gerado na saída, e sim o conteúdo do bloco "*recover block*".

#visit, #recurse, #fallback	<#visit *node* using *namespace*> or <#visit *node*> <#recurse *node* using *namespace*> or <#recurse *node*> or <#recurse using *namespace*> or <#recurse> <#fallback>	Utilizado para processamento recursivo, na prática utilizado em conjunto com XML.

Quadro 3.1 - Diretivas da linguagem de *template* do FreeMarker
Fonte: Do próprio autor, adaptado de FreeMarker (2011)

Além desta linguagem de *template*, o FreeMarker possui outros elementos importantes, que são usados em conjunto com a linguagem e aumentam a sua atuação: expressões, funções *built-in* e macros. A documentação online do FreeMarker apresenta em

detalhes e com vários exemplos todos os elementos da linguagem de *template* e de todos os recursos que podem ser utilizados pelo componente.

3.2 Velocity Apache

O Velocity é um projeto da *Apache Software Foundation,* conhecida por criar e manter vários outros projetos de software *open-source.* Este *engine* é oferecido através da *Apache Software Licence*, que o libera para uso público gratuito. (VELOCITY, 2011).

O Velocity é um simples mas poderoso *template engine* que renderiza dados a partir de objetos Java, conta com uma linguagem de *template* que permite, por exemplo, a desenvolvedores Web obter de uma forma fácil a apresentação de informações dinâmicas para usuários de um *site* ou de uma aplicação. Apesar do Velocity ter sido desenvolvido com a finalidade de ser utilizado em aplicações Web é possível também gerar vários outros formatos de saída, tais como: texto, XML, *e-mail*, SQL, *PostScript*, HTML, entre outros. A sintaxe dos *templates* e a arquitetura da ferramenta são fáceis de entender e rápidas de aprender e implementar (GRADECKI e COLE, 2003).

Este *template engine* também possui uma linguagem própria para definição dos *templates* e integração com uma coleção de classes Java que são usadas para criar uma ponte entre os componentes de modelo e de visão em uma arquitetura MVC (*Model-View-Controller*). Um dos melhores recursos do Velocity é a clara separação entre a visão e o resto do paradigma provendo somente um conjunto simples de sintaxe, no qual um *designer* Web usa para mostrar o conteúdo de uma aplicação. Ao mesmo tempo, programadores Java concentram-se na lógica da aplicação. A linguagem de definição de *templates* do Velocity não serve somente para desenvolvimento de páginas Web, mas também para todos aqueles que criam qualquer tipo de aplicações *standalone*. A saída do código gerado pelo Velocity pode ser HTML ou qualquer outro conteúdo como código SQL, XML, ou qualquer outra linguagem de programação ou mesmo uma DSL (*Domain-Specific Language*, ou linguagem específica de domínio) (GRADECKI e COLE, 2003).

Segundo Gradecki e Cole (2003), alguns dos principais componentes e recursos do Velocity são:

- Uma completa linguagem para manipulação de conteúdo de *templates* incluindo *loops* e condicionais;
- Acesso direto a métodos em objetos Java;
- Suporte direto ao Apache Turbine Application *Framework*;

- Transformação de XML para outros conteúdos usando Anakia (subprojeto do Velocity);
- Suporte ao Texen, que é um utilitário de geração de texto de propósito genérico;
- Suporte direto ao uso em *Servlets*.

3.2.1 Velocity Template Language

Assim como FreeMarker, o Velocity também apresenta uma linguagem de *template* com vários elementos. Embora esta linguagem não seja tão ampla quanto a do primeiro *template engine* apresentado, todos os elementos básicos são bastante parecidos, e permitem a construção de lógicas bastante complexas para elaboração de modelos de código fonte. O Quadro 3.2 apresenta as diretivas da *Velocity Template Language*.

Diretiva	**Sintaxe**	**Propósito**

`#foreach`	`#foreach ($item in $collection)`
	`  item is $item`
	`#end`
		Iterar sobre uma estrutura de dados *collection, array* ou *map*.

`#if`
`#else`
`#elseif`
	`#if ($order.total == 0)`
	`  No charge`
	`#end`
		Condicional. Nulos podem ser testados usando-se `#if ($obj) obj not null #end` syntax.

`#parse`	`#parse("header.vm")`	Carrega o *template*, executa o *parser*, e incorpora o *template* específico na saída gerada.
`#macro`	`#macro(currency $amount)${formatter.currency($amount)}#end`	Define uma nova diretiva e quaisquer parâmetros requeridos. O resultado é interpretado quando usado posteriormente no *template*. O uso da macro do exemplo seria: `#currency($item.cost)`.
`#include`	`#include("disclaimer.txt")`	Inclui o conteúdo do arquivo como ele é, na saída gerada.
`#parse`	`#parse("me.vm")`	Interpreta o *template* especificado e inclui a sua

Diretiva	Sintaxe	Propósito
		saída no código gerado.
#set	#set ($customer = ${order.customer})	Assinala um valor para um objeto de contexto. Se o objeto de contexto não existe, ele é adicionado, caso contrário é substituído.
#stop	#if ($debug) #stop #end	Pára o processamento do *template*. Isto é utilizado para propósitos de depuração.

Quadro 3.2 - Diretivas da linguagem de *template* do Velocity
Fonte: Do próprio autor, adaptado de Apache Velocity (2011)

3.3 XDoclet

O XDoclet é um *framework* extensível, *metadata-driven* (dirigido à metadados) e *attribute-oriented* (orientado à atributos) para a geração de código fonte Java. Resumidamente XDoclet é uma ferramenta que executa a criação de código fonte, principalmente das partes repetitivas, deixando para o programador focar o desenvolvimento da lógica nas partes mais importantes da aplicação. A diferença entre o XDoclet e outros *frameworks* de geração de código é que as informações que são fornecidas ao XDoclet tem origem em marcações no código fonte existente com *tags* semelhantes às usadas para gerar documentação usando a ferramenta Javadoc (WALLS e RICHARDS, 2004).

A geração de código através do *template engine* do XDoclet se dá a partir da interpretação das *tags* inseridas em algumas classes da aplicação, da interpretação de arquivos com os *templates* (modelos) e geração do código fonte. Tudo isto é feito a partir Apache Ant, que é uma ferramenta de automatização de *build* de software Java. A Figura 3.3 demonstra o modelo de funcionamento do XDoclet.

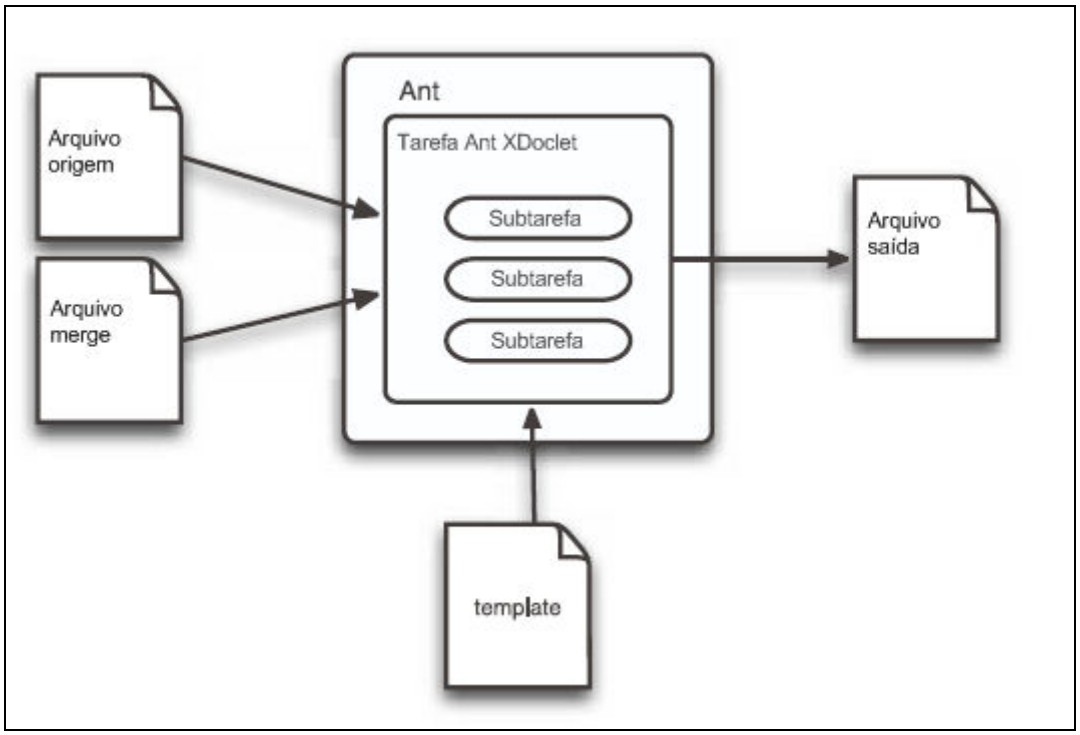

Figura 3.3 – Modelo de funcionamento do XDoclet
Fonte: Traduzido de Walls e Richards, 2004

Um dos problemas encontrados com a arquitetura do XDoclet, é que ele é bastante dependente de outras ferramentas. Ele é executado como um tarefa do Ant, ou seja, para que o mesmo seja executado, é preciso escrever um arquivo de *build* (*buildfile*), que chama as funcionalidades do XDoclet. Além disto, conforme citado por Herrington (2003), o gerador utiliza os comentários JavaDoc para chamar os módulos que criam uma grande variedade de arquivos de saída. O propósito inicial é a geração de *beans* de infraestutura, a partir de *entity beans* existentes. Embora seja possível a partir do código Java gerar vários artefatos em outras linguagens ou formatos, como XML, a metodologia utilizada pelo XDoclet é mais adequada para a geração de código fonte em Java, isto dificulta a sua utilização em uma ferramenta flexível que trabalhe com código gerado para várias linguagens de programação a partir de várias origens. Outro problema é que diferente dos outros dois *template engine*s estudados, para se ter maior flexibilidade de recursos de *template* boa parte do trabalho precisa ser feito em classes Java, através das chamadas *Tag Handlers*, que retorna as *strings* a inserir no código fonte gerado.

Devido as características mais específicas deste componente, o mesmo não foi tão explorado quanto os FreeMarker e Velocity. Mesmo assim, seus conceitos agregam no sentido de garantir que a abordagem é similar e que a separação da lógica e apresentação precisa ser separada.

3.4 Comparação dos *template engines*

A fim de definir qual é o melhor *template engine* para ser utilizado no gerador de código proposto neste trabalho, foi efetuada avaliação de várias características de cada um dos três estudados. Uma das principais características da comparação é a possibilidade de executar iteração sobre estruturas de dados permitindo a geração de vários elementos repetitivos em um programa. O processamento condicional também é importante, pois permite definir se um determinado trecho de código deve ou não ser gerado, permitindo que todos os trechos condicionais bem como as decisões de gerá-los ou não sejam escritos no próprio *template*, não sendo necessário alterar a ferramenta de geração de código.

Outros recursos como execução de funções ou macros, inclusão de outros *templates*, também foram analisados, com a finalidade de verificar a flexibilidade de uso do componente. Itens como bibliografia disponível, legibilidade do código escrito e facilidade de uso são importantes, pois são ligados diretamente à proposta da criação de uma ferramenta que seja facilmente expansível. O Quadro 3.3, mostra as características de cada um dos *template engines* analisados.

Característica	FreeMarker	Velocity	XDoclet
Iteração	SIM	SIM	NÃO
Processamento Condicional	SIM	SIM	NÃO
Macros	SIM	SIM	N/A
Incluir outros *templates*	SIM	SIM	SIM
Parser de *templates* incluídos	SIM	SIM	N/A
Plugins de edição	SIM	SIM	SIM
Executar código Java embutido	NÃO	SIM	SIM (via *Tag Handlers*)
Chamada de macros recursivas	SIM	NÃO	NÃO
Saída de *loops*	SIM	NÃO	NÃO
Independência de outras tecnologias	ALTA	MÉDIA	BAIXA
Lança exceções ao referenciar Parâmetros não declarados	SIM	NÃO	SIM

Característica	FreeMarker	Velocity	XDoclet
Bibliografia	NÃO	SIM	SIM
Legibilidade	MÉDIA	FÁCIL	MÉDIA
Facilidade de uso	MÉDIA	ALTA	BAIXA

Quadro 3.3 – Comparação de *template engines*
Fonte: Do próprio autor

3.5 Definição do *template engine*

Após a análise das características de cada um dos *template engines* estudados, ficou claro que para desenvolver o *framework* proposto neste trabalho, o componente deveria ser de uso flexível e expansível. Uma vez que o XDoclet não atende a estes requisitos, foi descartado.

Tanto o FreeMarker quanto o Velocity são dois componentes bastante respeitados, possuem muitos recursos em comum e apresentam farto material de consulta. No entanto, observa-se que atualmente o FreeMarker está tendo uma aceitação maior. Um dos fatos que comprova isto é que a IDE Netbeans desde a versão 6.0 adotou este componente ao invés do Velocity que era utilizado em versões anteriores.

A escrita dos *templates* FreeMarker segue uma sintaxe um pouco mais poluída que a do Velocity, pois usa marcações similares a *tags* XML. No entanto, é um aspecto apenas visual, que não implica em nenhum outro problema adicional, visto que as IDEs Java (Eclipse e Netbeans) possuem *plugins* que auxiliam a escrita dos *templates,* identificando seus elementos e promovendo a colorização de sintaxe.

Sobre a questão de desempenho de execução, onde o Velocity mostrou-se um pouco mais lento que o FreeMarker, pode ser suprida mantendo-se os objetos que fazem o *parser* do *template* carregados durante uma execução repetida do componente. Desta forma a diferença mantém-se apenas na carga inicial do componente.

O Velocity permite referências a propriedades não existentes, e não emite menhum tipo de advertência. Isto pode parecer uma facilidade, mas na verdade esconde alguns problemas, visto que em caso de erro na escrita do *template* o resultado seria a não execução de uma substituição de algum trecho do código esperada.

A execução de macros no FreeMarker também é mais apurada, possui mais opções, sendo mais poderosa. Algumas funcionalidades do FreeMarker como poder executar um *break* em um laço também aumenta suas possibilidades de uso. O fato do FreeMarker possuir mais recursos do FreeMarker implica em uma curva de aprendizado maior, mas existem bastante exemplos de uso do componente disponíveis na documentação oficial.

Qualquer um dos dois componentes atenderiam perfeitamente aos propósitos deste trabalho, inclusive o núcleo do gerador de código poderia tratar com mais de um deles, permitindo a execução de um ou outro de acordo com o projeto em execução. No entanto o FreeMarker foi escolhido para ser o *template engine* a ser usado principalmente pela questão de garantir um maior tratamento de erros bem como uma maior flexibilidade devido a uma linguagem de *template* ser mais completa.

4 VISÃO GERAL DA ARQUITETURA PROPOSTA

Nas próximas seções, de acordo com os objetivos deste trabalho, será detalhada a arquitetura proposta para a ferramenta de geração de código fonte que será desenvolvida, bem como a descrição de cada um de seus módulos.

4.1 Framework de geração de código

A proposta deste trabalho é a construção de um *framework* com padrão aberto que possa ser altamente acoplável a novas especificações de entrada. Este conceito parte da idéia de que é necessária uma arquitetura separada em módulos, com um núcleo bem definido que usa conceitos de geração de código fonte, de forma que possa haver uma dinâmica entre definições de entradas e saídas. Busca-se a resolução de problemas genéricos de geração de código, através do reaproveitamento de algumas definições de entrada comuns, tais como uma especificação padrão de XML, modelagem de dados, estrutura existente de um banco de dados qualquer e dicionários de dados de sistemas específicos. Pretende-se também que este *framework* possa ser utilizado para criar um gerador de código que possa ser usado em vários projetos, independente de linguagem de programação, sistema operacional, metodologia de desenvolvimento ou ferramentas utilizadas, reduzindo a incompatibilidade entre estas ferramentas que acabam prejudicando a produtividade das equipes de desenvolvimento de software.

Um ponto fundamental da arquitetura é que o núcleo de geração de código seja desvinculado das especificações das entradas. A proposta para resolver esta questão é através da definição de uma interface padrão que permite que novas especificações sejam escritas sem que seja necessário lidar com detalhes sobre o tratamento de *templates* (modelos de código), interação do usuário ou formatação de saída. O modelo proposto diferencia-se de outras ferramentas básicas de geração de código que não fazem o *merge* (combinação) de entradas com *templates*, bem como de soluções mais avançadas que requerem a implementação do relacionamento entre os componentes em um único módulo, dificultando ou impedindo a expansão da ferramenta, visto que são dependentes de origens e destinos muito específicos.

A solução proposta é estruturada na forma de módulos, que são: especificações de entrada, interface e núcleo. A ferramenta de geração de código desenvolvida utilizando o *framework* proposto permite acesso às suas funcionalidades por linha de comando e interface

gráfica, tendo como resultado final trechos de código fonte gerados de acordo com o especificado nos *templates*. A Figura 4.1 apresenta um diagrama resumido desta estrutura.

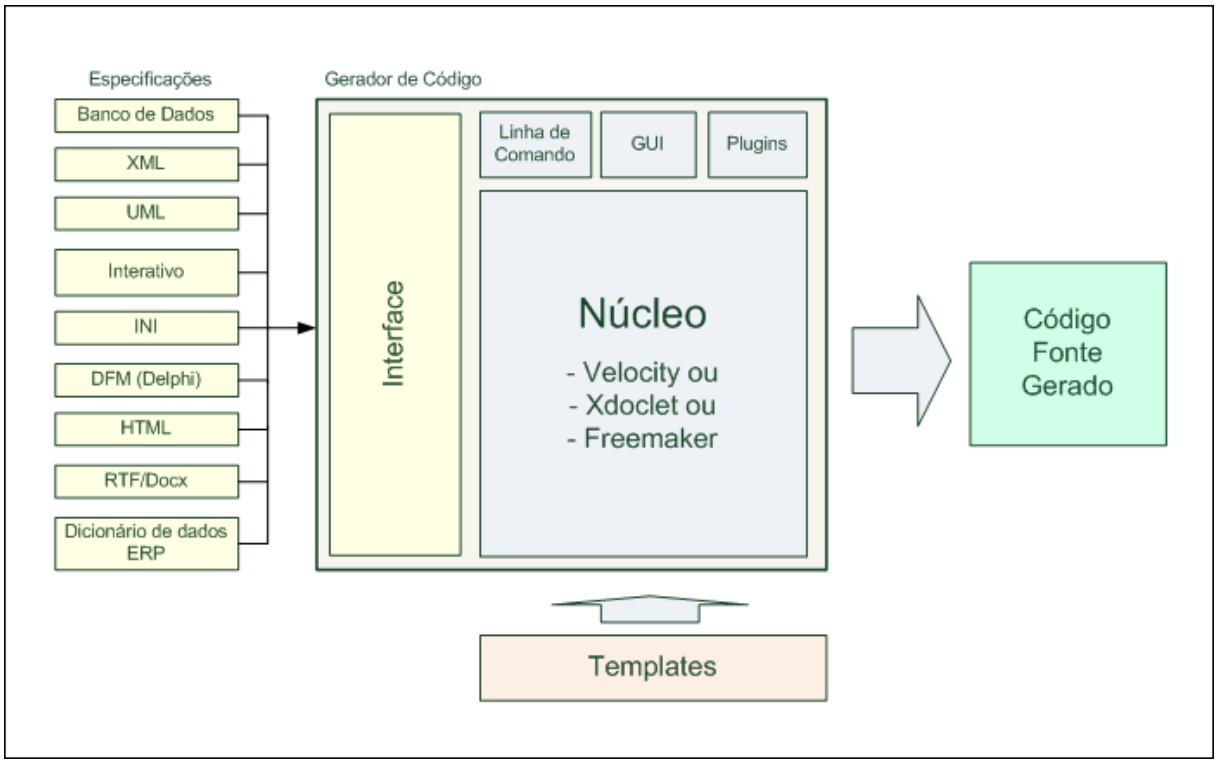

Figura 4.1 - Arquitetura da ferramenta de geração de código
Fonte: Do próprio autor

As próximas seções apresentam cada um dos módulos da ferramenta de geração de código, detalhando suas características principais.

4.2 Especificações de entrada

A geração de código fonte de qualquer artefato de software pela ferramenta proposta precisa se basear em uma origem, seja esta origem um repositório, os metadados de um banco de dados, um modelo conceitual ou uma especificação escrita em alguma linguagem de domínio específico. A ideia central deste trabalho, de permitir o desacoplamento das partes envolvidas em uma ferramenta de geração de código, prevê que para cada origem com estrutura distinta exista um programa que faça a interpretação da entrada e a entregue para um núcleo de processamento, que de forma padronizada faça o seu devido tratamento. Logo, uma especificação de entrada nada mais é que um programa escrito a partir de um modelo bem definido e padronizado que tem por objetivo colocar os dados de entrada em uma estrutura de dados que poderá ser tratada de uma forma única pelo núcleo de geração de código.

A interpretação de uma entrada pode ser feita de várias formas, desde um *parser* que utilize técnicas simples, até modelos bastante avançados que aplique o reconhecimento léxico e sintático de uma origem de entrada. Fowler (2010) cita algumas formas de fazer a interpretação de uma linguagem específica de domínio, também aplicável a algumas das especificações de entrada ora propostas:

- **Tradução dirigida a delimitadores** – é a forma mais simples para criar um *parser*, aplicável principalmente se a origem tiver um formato simples, por exemplo, um arquivo INI. O método de processamento é separar o texto em linhas e processar cada uma das linhas, extraindo os seus elementos a partir de delimitadores conhecidos. Para determinados elementos processados o *parser* entra em estados distintos. Esta forma de processamento pode também utilizar expressões regulares para fazer a validação e extração dos elementos;
- **Tradução dirigida à sintaxe** – quando a DSL tiver uma complexidade maior é necessário aplicar as mesmas técnicas de um compilador: análise léxica, sintática e semântica. Segundo Sebesta (2000), a análise sintática (*parsing*), é o processo de rastrear ou de construir uma árvore sintática para determinada *string* de entrada. No entanto, este processo é o que tem a curva de aprendizado e complexidade de implementação maior. Para garantir uma maior produtividade, é possível utilizar alguma ferramenta que a partir de uma gramática BNF seja criada automaticamente as rotinas que farão a interpretação da linguagem. É importante notar que este conceito também é o de geração de código fonte: escreve-se uma definição e obtém-se código fonte especializado nesta técnica. Na prática isto pode ser feito por várias ferramentas, entre elas o Javacc. Embora este método não seja utilizado neste trabalho, é importante citá-lo pois seria a melhor escolha para a interpretação por exemplo de trechos de código fonte de uma linguagem de programação ou mesmo de scripts SQL, entre outras aplicações;
- **Incorporar a DSL em uma linguagem existente** – muitas ferramentas de modelagem de dados geram seus modelos diretamente em XML, sendo que ferramentas de desenvolvimento distintas compartilham dados através deste formato. Sua interpretação pode ser facilitada em Java com o uso de componentes próprios para este fim tais quais: DOM, JDOM, JAXB e SAX. Este método é bastante prático e permite a carga de uma especificação que tenha origem, por exemplo, em modelos gerados pelo ArgoUML. Este método de

extração será utilizado na construção das especificações de XML e derivadas desta.

A possibilidade de expansão é o principal motivo de trabalhar com a interpretação separada como está sendo concebida. Desta forma, cada vez que surgir uma nova fonte de origem diferente das especificações já implementadas anteriormente pelo gerador de código, basta que seja escrita uma nova classe similar às existentes que trate a nova especificação, isso sem que nessa camada seja necessário ter qualquer tratamento da tecnologia usada para gerar código, apenas deve tratar de interpretar a estrutura de origem e transformar em uma árvore hierárquica que representa esta entidade para o fim de geração de código.

É possível elaborar até mesmo especificações mistas, que misturem mais de uma especificação, dependendo das necessidades do projeto a ser implementado. Por exemplo, poderia haver a leitura de uma estrutura vinda de um banco de dados e de informações adicionais extraídas de uma ferramenta de modelagem ou de um dicionário de dados de um sistema específico.

O papel do programa provedor da estrutura hierárquica (e especificação de entrada) é o de atuar sobre uma determinada origem de entrada, posicionando-se em um ponto específico e retornando sob demanda cada elemento e seus atributos. Por exemplo, no caso do banco de dados, o programa recebe o nome de um esquema e de uma tabela, procura pelo mesmo no banco de dados e retorna uma estrutura com os atributos referentes a cada campo, tais como nome, tipo de dado, tamanho, domínio, descrição detalhada, etc. Tais atributos são usados então pelo núcleo de geração de código para unir aos *templates* criando o código fonte. Como resultado, a linguagem do *template engine* pode percorrer estas estruturas permitindo gerar código, por exemplo, para cada atributo de uma entidade de um modelo de dados.

O armazenamento interno dos dados carregados pela especificação de entrada precisa ser feito de forma padronizada, porque novas especificações precisam estar no mesmo formato. Para tal organização, foi definido que a forma mais flexível e prática para manter estas estruturas seria a utilização de classes de coleções encontradas na API do Java. Desta forma, os dados da especificação de entrada foram armazenadas em Map (LinkedHashMap) e List (ArrayList), permitindo representar os nodos de uma árvore. Estas árvores podem ter vários níveis, visto que é possível colocar um nodo dentro de outro em sucessivos níveis de hierarquia. Isto é necessário porque a maioria das origens de dados que serão transformados em código fonte seguem uma hierarquia. A maioria pode ser expressa em alguns poucos níveis hierárquicos, como um arquivo INI ou uma estrutura de banco de dados, mas

modelagens mais complexas podem ter vários níveis, é o caso por exemplo, de um arquivo XML contendo um modelo de dados ou de classes.

Para efetuar o caminhamento pelas estruturas de dados foram desenvolvidos métodos *iterators* (iteradores). De acordo com Goodrich e Tomassia (2002), um *iterator* é um padrão de projeto de software que abstrai o processo de busca sobre uma coleção de elementos, um de cada vez. Um *iterator* consiste de uma sequencia S, uma posição corrente sobre S e uma forma de avançar para a próxima posição em S, tornando-se a posição corrente. Este caminhamento nas estruturas foi usado também na rotina de exibição dos dados em forma de árvore, no qual a estrutura da especificação é mostrado na interface de linha de comando.

Um exemplo de como é a organização da árvore hierárquica é mostrado no Quadro 4.1. Analisando-se o quadro é possível verificar que tanto a raiz quanto os nodos podem conter várias entradas no estilo chave/valor. A escolha por usar entradas nesse formato é porque elas garantem muita flexibilidade, e permitem representar vários níveis de hierarquia, também não há limitações de tamanho ou tipos de dados a serem usados, isto também garante uma grande facilidade para criar novas especificações, respeitando suas estruturas originais. A chave é sempre identificada por um conteúdo String (texto), por exemplo **atributoGeral1**, mostrado no mesmo quadro, no entanto os valores de cada entrada podem ser de vários tipos, tais como inteiro, texto, data e também outras listas, o que permite colocar nodos dentro de outros nodos, por exemplo, **Característica3**, mostrada dentro de **Nodo2**, poderia ser um **Nodo3**, e assim sucessivamente sem limites de número de níveis. Cada elemento que pode ter repetições que serão iterados nos *templates* devem ser colocados em uma lista, porque desta forma estarão disponíveis para serem acessados item a item expondo suas características em variáveis individuais.

<table>
<tr><td colspan="2">Raiz : LinkedHashMap</td></tr>
<tr><td>atributoGeral1</td><td>Conteúdo1 : dado do tipo String</td></tr>
<tr><td>atributoGeral2</td><td>Conteúdo2 : dado do tipo inteiro</td></tr>
<tr><td>atributoGeral3</td><td>Conteúdo3 : dado do tipo Date</td></tr>
<tr><td>Nome da Lista de atributos</td><td>NodosAtributos : Arraylist
<table>
<tr><td colspan="2">Nodo1 : LinkedHashMap</td></tr>
<tr><td>Característica1</td><td>"a"</td></tr>
<tr><td>Característica2</td><td>"b"</td></tr>
<tr><td>Característica3</td><td>"c"</td></tr>
</table>
<table>
<tr><td colspan="2">Nodo2 : LinkedHashMap</td></tr>
<tr><td>Característica1</td><td>"d"</td></tr>
<tr><td>Característica2</td><td>"e"</td></tr>
<tr><td>Característica3</td><td>"f"</td></tr>
</table></td></tr>
</table>

Quadro 4.1 – Estrutura de dados da especificação de entrada
Fonte: Do próprio autor

4.3 Interface

Para a construção das especificações de entrada, é importante que cada um dos programas distintos que tratam de traduzir uma estrutura específica em um mesmo modelo de dados reconhecido pelo núcleo de geração de código, implementem os mesmos métodos, ou seja, podem se comunicar com o núcleo da mesma forma, mesmo que retornem conteúdos com fins distintos, mas ainda assim de uma forma padronizada. Este contrato de implementação rege as regras que devem ser seguidas em cada programa que trate das especificações de entrada. Gamma et al (2000, p. 33), diz “Programe para uma interface, não

para uma implementação.", com isto, é possível reduzir as dependências de implementação e consiste em um importante princípio de projeto reutilizável orientado à objetos.

A organização das classes de especificações de entrada foram projetadas usando-se o conceito de herança e interfaces. A herança foi aplicada no desenvolvimento de todas as especificações de entrada desenvolvidas, sendo que elas reaproveitam os métodos de uma classe de especificação básica, a qual implementa comportamentos comuns a todas. Já o conceito de interface foi aplicado para garantir que todas as classes de especificações de entrada implementem comportamentos próprios, mas com os mesmos nomes e assinaturas de método, visto que o núcleo de geração de código irá chamá-los de forma padronizada. Para Arnold, Gosling e Holmes (2007), este conceito é considerado um contrato, ou seja, o comportamento atende à critérios específicos, mas os detalhes de implementação são irrelevantes.

Esta abordagem é importantíssima porque garante a uniformidade das especificações de entrada e visa a expansibilidade da ferramenta de geração de código e da consistência de sua arquitetura, foi desenvolvido desta forma por entender que "O principal elemento estrutural de Java que garante uma API é a **interface**." (GOODRICH e TOMASSIA, 2002, p 83). Na Figura 4.2 é mostrado o diagrama de classes de especificação básica, da interface para implementação das especificações e de duas classes distintas de especificações de entrada que foram desenvolvidas de acordo com este padrão, ressaltando que todas especificações herdam de uma especificação base e implementam uma interface que obriga a existência de alguns métodos.

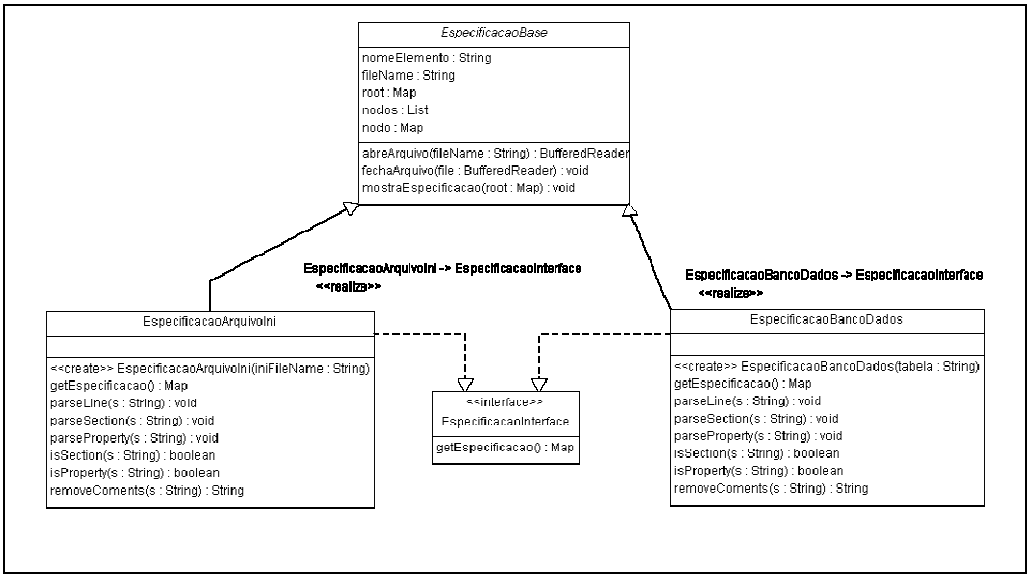

Figura 4.2 - Diagrama de classes das especificações de entrada com interface
Fonte: Do próprio autor

4.4 Núcleo

O núcleo de geração de código é a parte central da ferramenta, e integra todos os demais componentes da arquitetura proposta, dedicando-se as seguintes tarefas principais:

- **Interpretar a linha de comando ou projeto –** receber a linha de comando passada ao programa, separá-la em partes, interpretar seu conteúdo e validar cada uma das opções, bem como fornecer mensagens para o usuário;
- **Carregar e executar a especificação de entrada** – carregar o módulo responsável por fazer a análise da estrutura de entrada, bem como executar esta análise na parte de estrutura desejada pelo usuário;
- **Carregar o *template engine*** – a carga do componente que irá tratar o *template* bem como suas configurações devem ser uma das primeiras etapas;
- **Processar o template** – caso o *template* citado não esteja disponível para ser processado ou algum outro problema ocorrer deverá fornecer mensagens sobre estas situações;
- **Executar a geração de código** – esta etapa também pode ser chamada de *merge*, ou seja, a função de mesclar dados de entrada com o modelo de código, criando em um *buffer* de memória o código fonte de saída;
- **Criar os arquivos de código fonte gerados** – transferir o código gerado para uma localização física, tratando de sua gravação, bem como configurações de projetos necessários para sua utilização pelas ferramentas de desenvolvimento.

4.5 Linha de comando

A ferramenta terá um módulo responsável pela identificação da linha de comando para a chamada da ferramenta de geração de código. A importância da existência de uma interface por linha de comando é a de possibilitar a criação de scripts para gerar vários artefatos de software de forma automatizada, permitindo por exemplo que ao alterar-se um *template*, sejam gerados novamente todos os programas que seguem o mesmo modelo. As seguintes opções básicas estarão disponíveis:

- **Opção –e (entrada) –** utilizada para informar ao gerador de código qual especificação de entrada deve ser utilizada. Seu nome deve ser o mesmo da classe a ser carregada e executada para interpretação dos dados de origem;

- **Opção –n (nome) –** opção utilizada para informar ao gerador de código qual o nome do elemento existente na origem será utilizado para geração de código;
- **Opção –t (template) –** nome do *template* que deverá ser utilizado para geração do código fonte, também poderá indicar o caminho do arquivo;
- **Opção –s (saída) -** caminho e nome do arquivo de saída o qual deverá se usado para gravar o código fonte gerado;
- **Opção –ees (exibe especificação de entrada) –** após a carga da especificação de entrada, percorre e mostra sua estrutura hierárquica, com cada elemento que foi carregado e seus atributos (pares chave/valor).

Exemplo de linha de comando para execução do gerador de código:

```
Java –jar Gerador.jar –e=XML –n=Produto –t=Bean.ftl –s=Produto.java
```

4.6 Interface gráfica de usuário

A interface gráfica de usuário do gerador de código permite a especificação de todos os parâmetros que poderão ser informados via linha de comando, bem como auxiliar com diálogos de abertura de arquivo na localização de pastas e arquivos. Também deverá permitir a visualização dos *templates* e do código gerado.

Para facilitar a configuração de um conjunto de artefatos a serem gerados, foi criada uma interface gráfica com o usuário na forma de *wizard* (assistente). O objetivo é solicitar passo a passo cada opção e no final além de gerar o código fonte esta configuração é gravada em um arquivo de projeto, o qual pode ser reutilizado no futuro para sucessivas gerações de código fonte.

4.7 *Templates*

Os *templates*, escritos na linguagem do *template engine* FreeMarker deverão ser validados e interpretados para processamento. É importante que os *templates* sempre fiquem em um local físico diferente dos demais elementos utilizados no gerador de código, preferencialmente devem residir junto a seus projetos de origem ou ao menos em uma pasta separada.

4.8 Saídas

As saídas do gerador de código são arquivos texto em codificação idêntica aos *templates*, e devem ser armazenados onde for indicado na linha de comando ou pela interface gráfica da ferramenta. Esta é parte importante que precisa da garantia de que haverá um bom tratamento das rotinas de gravação e tratamento de erros de exceção.

4.9 Ferramentas de desenvolvimento utilizadas

Para desenvolver o *framework* e a ferramenta de geração de código foram escolhidas algumas ferramentas de desenvolvimento de padrão aberto e de uso bastante difundido, permitindo uma futura expansão de forma bastante fácil para o uso em outros projetos. Algumas das ferramentas são usadas tanto para desenvolver o gerador de código quanto para sua utilização em um projeto específico, seja como ferramenta de análise e modelagem ou como suporte para execução.

- **Java 7 -** Java é uma plataforma base para o desenvolvimento de praticamente qualquer tipo de aplicativo em rede e padrão global para desenvolvimento e fornecimento de aplicativos para celular, jogos, conteúdo on-line e software corporativo;
- **Netbeans 7.1** - é uma IDE, ou seja, um ambiente integrado de desenvolvimento multiplataforma, podendo ser executado em Windows, Linux e Solaris. Consiste em uma plataforma que permite o desenvolvimento de aplicações Web, *desktop*, *enterprise* e *mobile* usando a plataforma Java, bem como em outras linguagens, tais quais PHP, Ajax, Groovy and Grails e C/C++. Netbeans suporta também o desenvolvimento de *plugins* de terceiros, permitindo expandir suas funcionalidades;
- **ArgoUML 0.34 -** é uma aplicação *open source* para modelagem UML a qual inclui suporte para todo o padrão 1.4 de diagramas UML. A aplicação é executada em qualquer plataforma Java e está disponível em 10 idiomas. A ferramenta também permite a execução de engenharia reversa e geração de código Java e C#. O ArgoUML foi utilizado nesse trabalho para projeto da arquitetura da aplicação e também para a elaboração de exemplos práticos de modelos de classes que servem como origem para geração de código;

- **MySQL -** banco de dados muito popular pelo apelo *open source*, alta performance, simplicidade de instalação, configuração e utilização. É o banco de dados escolhido por toda uma nova geração de aplicações tais quais Apache, PHP, Perl, Python, entre outras. MySQL também executa em mais de 20 plataformas distintas e possui um conjunto de ferramentas flexíveis para o seu controle. Embora tenha sido utilizado o MySQL para este trabalho, qualquer outro banco de dados com acesso via JDBC poderia ser utilizado com resultado similar;
- **MySQL Workbench -** é uma ferramenta visual unificada para arquitetos de banco de dados, desenvolvedores e DBAs (*Data Base Administrators*). A ferramenta permite modelagem de dados, desenvolvimento SQL, além de configuração e administração de bancos de dados MySQL. A ferramenta é uma evolução do DBDesigner e é disponibilizado para Windows, Linux e Mac OS.

5 FUNCIONAMENTO DO GERADOR DE CÓDIGO

O objetivo deste capítulo é apresentar alguns casos de uso da ferramenta de geração de código executados a partir das especificações de entrada desenvolvidas, detalhando o seu funcionamento e comprovando a ideia de que é possível a criação de uma especificação para cada formato de entrada, expandindo em forma de novos módulos o uso da ferramenta.

Devido à estrutura ter sido concebida baseada em interface e herança, cada nova especificação de entrada conta com uma série de facilidades no que diz respeito à leitura de arquivos, estrutura de dados com modelo hierárquico em uma árvore bem definida e força a implementação de métodos padronizados para efetuar as tarefas necessárias. No entanto não impede-se que implementações específicas possam ser feitas para outros fins.

Serão apresentados casos de uso do gerador de código com especificações de entrada escritas para arquivos INI, estrutura de banco de dados, XML e dicionário de dados do SIGER. Cada seção terá a mesma estrutura, detalhando a origem dos dados, o formato da especificação desenvolvida, os *templates* utilizados e o resultado da geração de código.

5.1 Arquivo INI

Nesta seção será detalhado o funcionamento básico da ferramenta de geração de código, onde a partir da configuração de uma classe descrita em um arquivo INI são criadas classes básicas em linguagem Java.

Arquivos INI são usados como um formato padrão para arquivos de configuração. Embora este formato de arquivo tenha sido substituído pelo registro do Windows em muitas aplicações, outras tantas ainda o utilizam para guardar dados de configuração de uma forma simples que pode ser alterada tanto pela aplicação quanto por um editor de textos comum, desvinculando do sistema operacional e permitindo o uso de forma portável. Devido a sua estrutura simples, que permite especificar pares chave/valor dentro de seções, foi escolhido para construir a primeira especificação de entrada permitindo exemplificar o funcionamento básico do gerador de código implementado.

5.1.1 Formato da especificação

A interpretação do formato foi feita na forma de um *parser* desenvolvido baseado no algoritmo denominado "*Delimiter-Direct Translation*" (tradução direta por delimitador), o algoritmo foi proposto por Fowler (2010), que diz que este formato por não ter uma estrutura

tão complexa quanto outros formatos de arquivo tais como XML, é ideal para demonstrar o básico no tratamento de uma DSL. A Figura 5.1 mostra a estrutura de um arquivo INI, apresentando duas seções e suas propriedades.

```
;Comentário texto livre
[section1]
Propriedade1=valor1
Propriedade2=valor2

;Comentário texto livre
[section2]
Propriedade1=valor3
Propriedade2=valor4
```

Figura 5.1 – Estrutura de um arquivo INI
Fonte: Do próprio autor

5.1.2 Exemplo prático

Para entendimento do funcionamento básico da ferramenta de geração de código, toma-se por exemplo uma especificação de um objeto escrito em um arquivo INI. O nome do arquivo é o mesmo do objeto, as seções deste arquivo contém o nome de seus atributos, e as entradas (pares chave/valor), representam as características de cada atributo. O exemplo a seguir, (Figura 5.2) é o modelo básico de um livro, que foi definido no arquivo **livro.ini**, escrito em qualquer editor de textos padrão, com alguns dos atributos que este objeto teria para ser implementado em uma classe Java.

```
[titulo]
tipo = String
acesso = private
descricao = título do livro

[autor]
tipo = Autor
acesso = private
descricao = autor do livro

[editora]
tipo = Editora
acesso = private
descricao = editora do livro

[local]
tipo = String
acesso = private
descricao = local de edição

[ano]
tipo = int
acesso = public
descricao = ano de publicação

[paginas]
tipo = int
acesso = private
descricao = número de páginas

[disponivel]
tipo = boolean
acesso = private
descricao = se está disponível para empréstimo
```

Figura 5.2 – Modelo da classe “livro”
Fonte: Do próprio autor

A classe de especificação de entrada coloca os dados em uma estrutura hierárquica onde é possível verificar-se as características gerais do objeto bem como as características de cada atributo, a Figura 5.3 mostra esta estrutura.

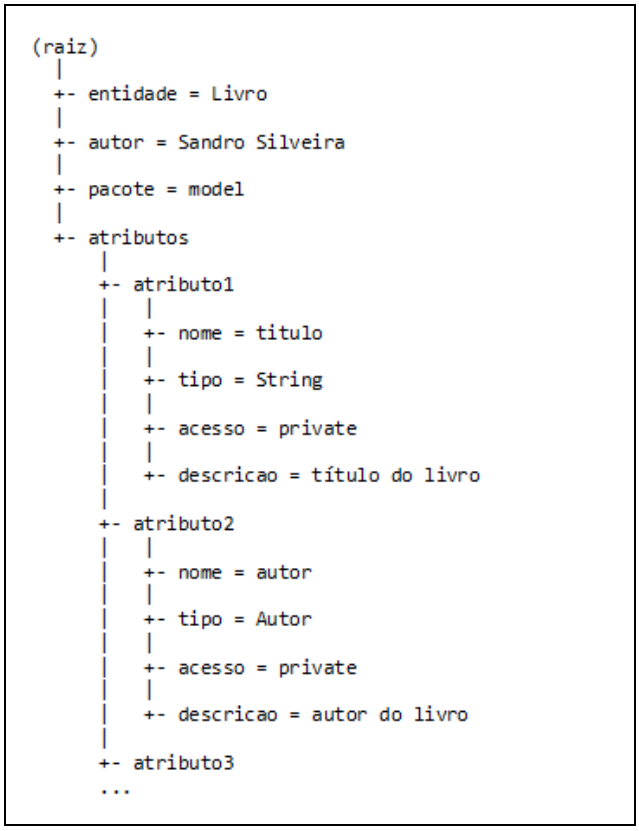

Figura 5.3 - Estrutura hierárquica dos dados da classe livro
Fonte: Do próprio autor

5.1.3 Template

Foi escrito um *template* que descreve a estrutura do código fonte a ser gerado. Para este exemplo será apresentado o código base para um POJO (*Plain Old Java Object*) ou *bean*, ou seja, um objeto Java que contém todos os seus atributos e algumas características padrão, como um construtor vazio. Definindo um objeto desta forma, pode-se utilizar a mesma especificação em diversos tipos de *framework* diferentes, visto que possui um padrão largamente difundido. A Figura 5.4 mostra como este *template* foi estruturado.

```
<#-- Template para geração de um objeto básico que encapsula os atributos de
uma classe -->
/*
 * Autor: ${autor}
 *
 */
package ${package};

public class ${entidade} {

<#list atributos as a>
    /**
     * ${a.descricao?cap_first}
     */
    ${a.acesso} ${a.tipo} ${a.nome};

</#list>
<#list atributos as a>
    /**
     * Seta ${a.descricao}
     */
    public void set${a.nome?cap_first}(${a.tipo} ${a.nome}) {
        this.${a.nome} = ${a.nome};
    }

    /**
     * Retorna ${a.descricao}
     */
    public ${a.tipo} get${a.nome?cap_first}() {
        return ${a.nome};
    }

</#list>
}
```

Figura 5.4 – Template básico para geração de uma classe POJO
Fonte: Do próprio autor

Diversos elementos da linguagem de *template* do FreeMarker foram utilizados neste exemplo, são eles: comentários, variáveis, listas e funções *built-in*.

Comentários – todos os trechos inseridos entre as *tags* “`<#--`“ e “`-->`” são comentários apenas para documentar o *template*, ou seja, não são transferidos para o código fonte gerado.

Variáveis – armazenam os valores que são colocados nas especificações de entrada e utilizados pelo núcleo de geração de código para mesclar com os *templates* gerando o código final. São identificados entre as *tags* “`${`“ e “`}`”. Para estruturas com mais de um nível de hierarquia, podem ser acessadas por uma convenção que utiliza o nome da estrutura seguida de um ponto e o nome da variável.

Listas – permite iterar entre elementos de uma sequência, sendo definidas com as *tags* “`<list (nome_lista) as (nome_item)>`” e “`</#list>.` Esta estrutura executa um laço que percorre todos os elementos, colocando cada item da lista disponível para geração de código por vez.

Funções *Built-in* – diversas funções de texto, número, data, entre outras estão disponíveis na linguagem do FreeMarker. Também é possível a criação de outras funções personalizadas, tanto dentro do próprio *template* quanto escrita em código Java, sendo nesse caso necessário fazer a ligação pelo núcleo do gerador de código. As funções usadas no exemplo são escritas após o caracter de "`?`" dentro da *tag* de definição de variável, como em "`get${a.nome?cap_first}()`", que provê uma maneira de passar a primeira letra do conteúdo da variável para maiúscula.

Com a especificação de entrada e o *template* construídos, é possível utilizar o gerador de código. A partir do *prompt* deve-se digitar a linha de comando mostrada na Figura 5.5, onde constam as opções informadas para a geração do programa **Livro.java**, conforme parâmetros detalhados no capítulo 4.

```
Java  -jar Gerador.jar -e=ArquivoIni -n=Livro -t=pojo3.ftl -s=Livro.java
```

Figura 5.5 – Linha de comando completa para geração de código
Fonte: Do próprio autor

A opção "`-e=ArquivoIni`" indica que a especificação de entrada de arquivos INI será utilizada para alimentar as estruturas de dados para geração de código. Na sequência a opção "`-n=Livro`" determina o nome do elemento que será usado para geração de código, no caso desta especificação será usado o arquivo **Livro.ini**. Já a opção "`-t=pojo3.ftl`" especifica o *template* que é usado para geração de código. Finalmente a última opção descrita como "`-s=Livro.java`" irá determinar o nome do arquivo de saída.

5.1.4 Resultado da geração de código

A Figura 5.6 mostra trecho do código fonte gerado. O código fonte completo está disponível no APÊNDICE A. Nota-se que além dos elementos necessários do programa há o cuidado de gerar um código fonte totalmente legível, com identação correta e com comentários seguindo um padrão de codificação pré-determinado. Este padrão segue as orientações de Fowler (2010), que explica que há duas escolas de pensamento sobre código fonte gerado. Uma que diz que se o código não deve sofrer manutenção não precisa haver uma preocupação com a sua formatação. A outra indica que o código seja bem estruturado e limpo, visto que pode ser usado para depuração e documentação. Esta última é sua preferência pois torna o código mais legível mesmo que eventualmente possa ocorrer algumas duplicações de informação como os comentários. A documentação é parte importante do

código gerado. Kreisig (2007), explica que o gerador de artefatos do framework jFace também gera código fonte documentado com as descrições informadas no modelo para as entidades e atributos tornando possível a criação de JavaDoc.

```
/*
 * Autor: Sandro Silveira
 *
 */
package model;
public class Livro {
    /**
     * Título do livro
     */
    private String titulo;
    /**
     * Autor do livro
     */
    private Autor autor;

    ...

    /**
     * Seta título do livro
     */
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    /**
     * Retorna título do livro
     */
    public String getTitulo() {
        return titulo;
    }

    /**
     * Seta autor do livro
     */
    public void setAutor(Autor autor) {
        this.autor = autor;
    }

    /**
     * Retorna autor do livro
     */
    public Autor getAutor() {
        return autor;
    }

    ...
}
```

Figura 5.6 – Trecho de código fonte gerado
Fonte: Do próprio autor

5.2 Banco de dados

Nesta seção será apresentado um exemplo de geração de código tendo origem na estrutura de um banco de dados relacional. A partir desta estrutura será gerado código fonte para a persistência de dados.

Uma das especificações de entrada mais importantes para a ferramenta proposta é a que carrega a estrutura a partir de um banco de dados, visto que o resultado de uso desta especificação de entrada é permitir a geração de código fonte a partir de um banco de dados existente, tenha sido ele criado a partir de ferramentas de modelagem ou mesmo através do processo normal de desenvolvimento. Como resultado pode-se obter artefatos, por exemplo, referente à camada de persistência de dados, como um modelo DAO (*Data Acess Objects*). A Figura 5.7 mostra a hierarquia dos metadados que são utilizados para carga da especificação.

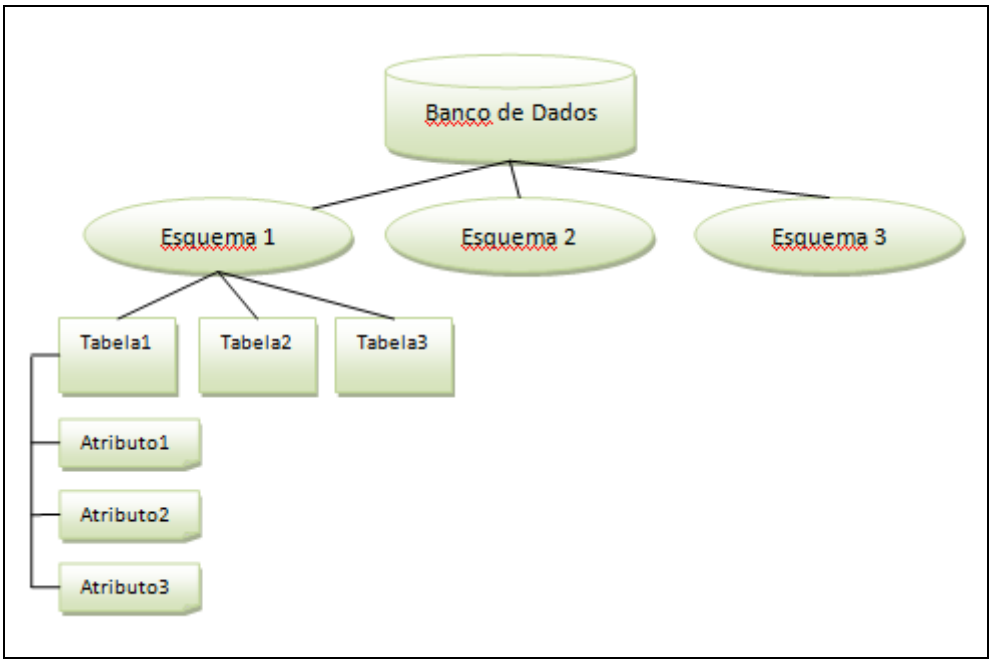

Figura 5.7- Hierarquia dos metadados de um banco de dados
Fonte: Do próprio autor

5.2.1 Formato da especificação

Para implementar este componente da ferramenta de geração de código, foram utilizadas as classes de acesso à metadados implementadas na API do Java que seguem o padrão JDBC. Com esta técnica é possível a partir de uma conexão com um banco de dados qualquer, seja ele Oracle, SQL-Server, MySQL, PostgreSQL entre outros, extrair informações do catálogo sobre esquemas, tabelas e atributos. A partir daí estas informações são

transferidas para as estruturas de dados que são utilizadas pelo núcleo, disponibilizando-as para que possam ser referenciadas nos *templates*.

Um dos itens importantes desta especificação é a compatibilização dos tipos de dados do banco de dados com os do Java. Para tal foi prevista a definição de uma tabela de compatibilidade, que para cada atributo encontrado seja relacionado com o devido tipo de dados correspondente. Foi escrita uma tabela de conversão para cada tipo de banco de dados utilizado com a ferramenta.

No Quadro 5.1 é apresentada a compatibilidade entre os tipos de dados, definidos para o banco de dados MySQL, que foi implementado para os exemplos práticos deste trabalho.

Tipo do banco de dados	**Tipo Java**
BIT	java.lang.Boolean
TINYINT	java.lang.Integer
BOOL, BOOLEAN	java.lang.Boolean
SMALLINT [UNSIGNED]	java.lang.Integer
MEDIUMINT [UNSIGNED]	java.lang.Long
INTEGER [UNSIGNED]	java.lang.Long
BIGINT [UNSIGNED]	java.math.BigInteger
FLOAT	java.lang.Float
DOUBLE	java.lang.Double
DECIMAL	java.math.BigDecimal
DATE	java.sql.Date
DATETIME	java.sql.Timestamp
TIMESTAMP	java.sql.Timestamp
TIME	java.sql.Time
YEAR	java.sql.Short
CHAR	java.lang.String
VARCHAR	java.lang.String
BINARY	byte[ ]
VARBINARY	byte[ ]
TINYBLOB	byte[ ]
VARCHAR	java.lang.String
BLOB	byte[ ]
VARCHAR	java.lang.String
MEDIUMBLOB	byte[ ]
VARCHAR	java.lang.String
LONGBLOB	byte[ ]
VARCHAR	java.lang.String
CHAR	java.lang.String
CHAR	java.lang.String
VARBINARY	byte [ ]
TINYBLOB	byte[ ]
VARCHAR	java.lang.String
CHAR	java.lang.String
CHAR	java.lang.String

Quadro 5.1 – Compatibilidade de dados do banco de dados x Java
Fonte: Documentação on-line MySQL

5.2.2 Exemplo prático

Para a utilização da geração de código fonte a partir de bancos de dados existentes, foi utilizado um modelo básico de um sistema de faturamento, baseado em um banco de dados de exemplo que acompanha a instalação do MySQL, denominado “AffableBean”. Este modelo foi traduzido e expandido para demonstrar a geração da camada DAO das tabelas existentes neste banco de dados. O código fonte gerado permite acessos de leitura, gravação e deleção nas tabelas, facilitando a construção desta camada para a aplicação em sistemas novos ou existentes. Também é possível a partir de sucessivas melhorias nos *templates*, a implementação de otimizações importantes como *pooling* de conexões, gerenciamento de *cache* e aplicação de técnicas de *lazy load* (um padrão de projeto conhecido por permitir a carga de objetos somente quando os mesmos sejam necessários).

Outra vantagem que a geração de código oferece é a criação de diferentes abordagens para atender a mesma necessidade, permitindo aos desenvolvedores facilmente compararem o desempenho de cada uma destas abordagens. Para o exemplo desse trabalho, a criação da camada DAO será com a utilização direta de JDBC. No entanto, apenas com a definição de outros *templates* é possível criar por exemplo, classes com *anotations* no padrão JPA (*Java Persistence API*), permitindo que o acesso ao banco de dados seja via TopLink ou Hibernate. A possibilidade de poder oferecer vários resultados distintos para uma mesma necessidade demonstra a flexibilidade e expansibilidade que é possível com a ferramenta de geração de código proposta neste trabalho.

O modelo de dados utilizado fornece o básico para a implementação de um sistema de faturamento ou vendas, possuindo tabelas que armazenam dados de clientes, produtos, categorias de produtos, pedidos e itens dos pedidos. Foram colocados alguns tipos de dados distintos nestas tabelas tais como inteiros, datas, decimais e texto, para demonstrar como os diferentes tipos de dados serão tratados na carga da especificação de entrada e do resultado final de código gerado para interagir com os mesmos. A Figura 5.8 mostra o diagrama das tabelas utilizadas nesta modelagem como origem para a geração de código.

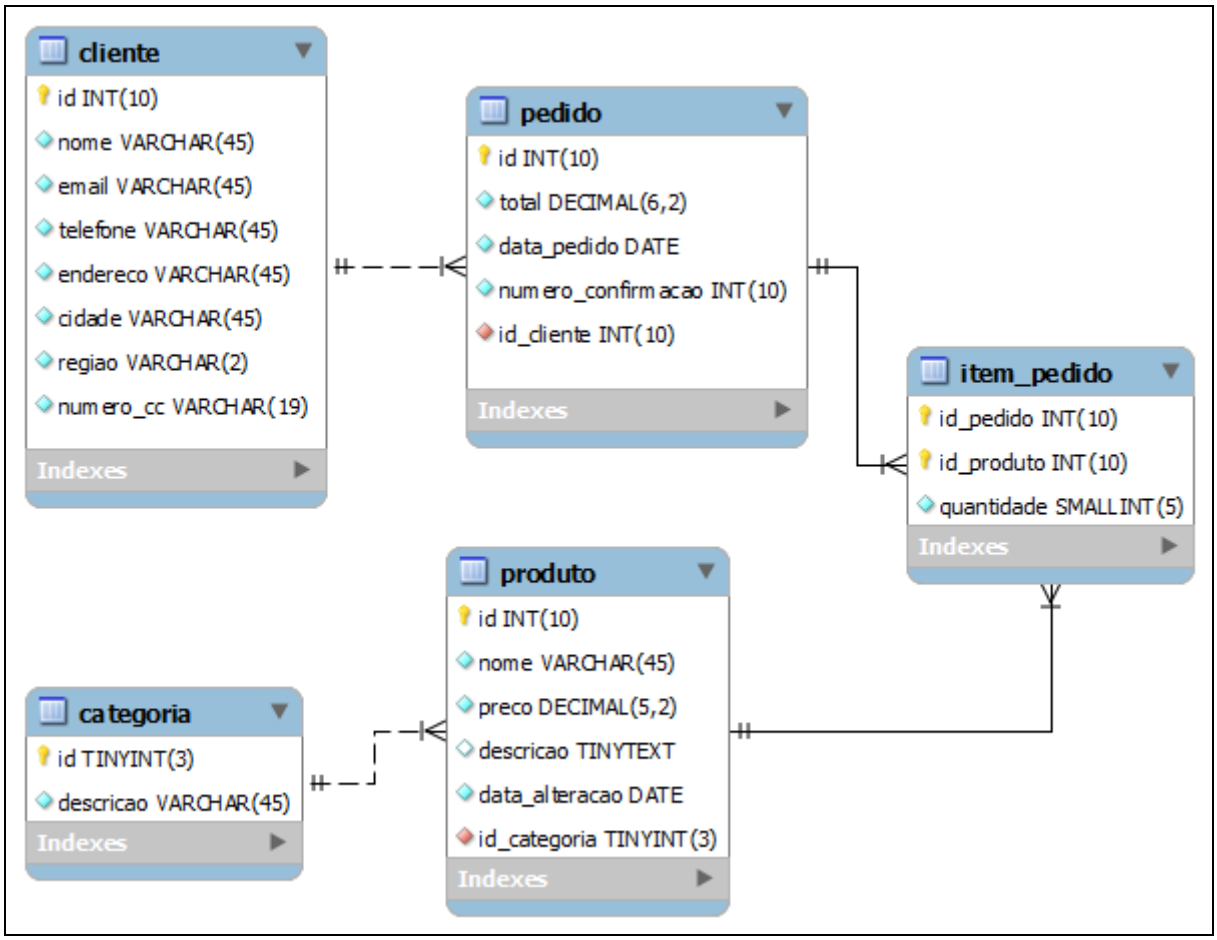

Figura 5.8 – Diagrama de tabelas usadas para geração da camada DAO
Fonte: Do próprio autor, adaptado do modelo "Affablebean" do MySQL

A especificação de entrada de banco de dados coloca na estrutura de dados diversas informações relativas à cada atributo. Além das características básicas utilizadas no exemplo inicial de arquivo INI, neste novo exemplo são necessárias mais informações para gerar código fonte de maior complexidade que o anterior. No Quadro 5.2 são relacionadas todas estas características que estão disponíveis para serem utilizadas nos *templates* para geração de código fonte podendo ser referenciadas como variáveis.

Variável	Tipo	Detalhamento
Nome	String	Nome do atributo, já compatibilizado para o padrão Java, sem caracteres como underline e com a letra inicial de cada palavra em maiúscula exceto a primeira.
Descrição	String	Descrição do atributo, texto que será utilizado nos *templates* para a criação de comentários e Javadoc, deixando o código fonte o mais compreensível e autodocumentado possível.
Coluna	String	Nome da coluna no banco de dados, utilizado para a montagem de sentenças SQL nos *templates*.
tipoBanco	String	Tipo de dados original do banco de dados, pode ser utilizado para a montagem de sentenças de definição de dados (DDL)

		caso os *templates* façam também a criação das tabelas.
isPk	Boolean	Indica que o atributo faz parte da *primary key* (chave primária) da tabela, utilizado nos *templates* para identificar estes atributos, e gerar código referente à montagem e movimentação de dados de/para chaves primárias.
Acesso	String	Tipo de acesso aos atributos das classes criadas nos *templates*, para esta especificação por *default* é gerado como *private,* no entanto outra especificação de entrada, por exemplo uma que carregasse de um modelo de classes UML seria o acesso definido pela arquitetura, conforme declarado no diagrama.
Nulos	Boolean	Indica se o atributo permite conter nulos;
Tipo	String	Tipo de dados Java, conforme compatibilização que foi citada no Quadro 5.1 – Compatibilidade de dados do banco de dados x Java
metodo_set	String	Método da API JDBC utilizado para transferir o dado.
objeto_compat	String	Classe que o tipo de objeto é compatível com um tipo primitivo, por exemplo, se o atributo for criado como int em Java, a classe que é compatível é Integer.
tamanho	Integer	Tamanho do atributo.
decimais	Integer	Número de casas decimais quando aplicável.

Quadro 5.2 – Variáveis usadas nos templates
Fonte: Do próprio autor

Um trecho da estrutura hierárquica dos dados que esta especificação de entrada carrega para a tabela de itens do pedido pode ser vista na Figura 5.9. Esta visão dos dados de origem carregados em forma de árvore é importantíssimo para o entendimento e facilita a elaboração dos *templates*.

```
(raiz)
  |
  +- entidade = ItemPedido
  |
  +- autor = Sandro Silveira
  |
  +- package = model
  |
  +- tabela = ITEM_PEDIDO
  |
  +- atributos
      |
      +- atributo
      |   |
      |   +- nome = idPedido
      |   |
      |   +- descricao = Identificador do pedido
      |   |
      |   +- coluna = id_pedido
      |   |
      |   +- tipoBanco = INT UNSIGNED
      |   |
      |   +- isPk = true
      |   |
      |   +- acesso = private
      |   |
      |   +- nulos = false
      |   |
      |   +- tipo = int
      |   |
      |   +- metodo_set = setInt
      |   |
      |   +- objeto_compat = Integer
      |   |
      |   +- tamanho = 10
      |   |
      |   +- decimais = 0
      |
      +- atributo
          +- nome = idProduto
          ...
```

Figura 5.9 – Estrutura hierárquica dos dados de itens do pedido
Fonte: Do próprio autor

5.2.3 Templates

Para a criação da camada DAO foram criados e utilizados os seguintes *templates*:

- **Dao1.ftl -** especifica a estrutura da classe de acesso à dados, possui todos os métodos necessários para inclusão, alteração e deleção de dados;
- **DaoException.ftl -** especifica a classe de tratamento de erros para cada entidade;
- **Pk.flt** –especifica a estrutura da classe que permite encapsular os atributos da *primary key* de cada tabela;
- **Pojo4.flt** - usado como um modelo mais complexo que o POJO básico apresentado no exemplo anterior, implementa inclusive os métodos "toString()" e "equals()" de cada entidade. A geração automática de classes com estes métodos mostra o quão importante é a abordagem de geração de toda a classe visto que as IDEs como Eclipse e Netbeans possuem auxílio para gerar estes métodos quando o código é criado manualmente, porém o programador pode implementar um novo atributo na classe e esquecer de acrescentá-lo ao método equals(), ocasionando erro de execução ao comparar objetos de uma mesma classe, visto que não levará em consideração a comparação do novo atributo. A Figura 5.10 mostra o trecho do *template* que define estes métodos.

Também pode-se observar mais um recurso que foi explorado da *FreeMarker Template Language*, que permite testar o índice do elemento atual de uma lista e também se este elemento é ou não o último. A sintaxe usada no *template* é **"**`<if [alias_da_lista]_has_next>`**",** podendo ser visto em **"**`<#if a_has_next>`**",** onde "a" é o alias para a lista de atributos. Este recurso é muito útil, quando precisa-se separar todos os atributos da classe com delimitadores, tais como vírgulas, ou ainda fechar uma construção em bloco, sendo o separador colocado em todos os elementos exceto no último, que deve ser diferente dos demais.

```
</#list>
    @Override
    public String toString() {
        return "${entidade}{" +
<#list atributos as a>
<#if a_has_next>
                    "${a.nome}=" + ${a.nome} + ", " +
<#else>
                    "${a.nome}=" + ${a.nome} + "}";
</#if>
</#list>
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (obj == this) {
            return true;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final ${entidade} outro${entidade} = (${entidade}) obj;
<#list atributos as a>
        if (this.${a.nome} != outro${entidade}.${a.nome}) {
            return false;
        }
</#list>
        return true;
    }
```

Figura 5.10 – Trecho de *template* para geração de métodos toString e equals
Fonte: Do próprio autor

5.2.4 Resultado da geração de código

O resultado final da geração do código para acesso aos dados de uma aplicação são as classes da camada DAO, que são completamente utilizáveis, sem que seja necessário alterar nenhuma linha de código. Desta forma, permite-se que sejam geradas novamente a cada mudança no modelo do banco de dados, seguindo a metodologia considerada por Walls e Richards (2004) como geração de código ativo.

A Figura 5.11 mostra o método **toString** gerado na classe POJO **Cliente.java** a partir do merge entre o template apresentado anteriormente com os dados da especificação.

```
@Override
public String toString() {
    return "Cliente{" +
                "id=" + id + ", " +
                "nome=" + nome + ", " +
                "email=" + email + ", " +
                "telefone=" + telefone + ", " +
                "endereco=" + endereco + ", " +
                "cidade=" + cidade + ", " +
                "regiao=" + regiao + ", " +
                "numeroCc=" + numeroCc + "}";
}
```

Figura 5.11 – Método toString gerado
Fonte: Do próprio autor

A Figura 5.12 mostra o projeto aberto na IDE Netbeans com todas as classes criadas, no APÊNDICE B pode-se verificar a listagem das classes de acesso a dados para uma das tabelas do modelo utilizado como exemplo, no caso a tabela "Produto". O conjunto de classes geradas para esta tabela do banco de dados são: **Produto.java**, que contém a estrutura dos dados, **ProdutoDao.java** que descreve todos os métodos de acesso a dados, **ProdutoDaoException.java** que efetua o tratamento de erros e **ProdutoPk.java** que encapsula a *primary key* da tabela.

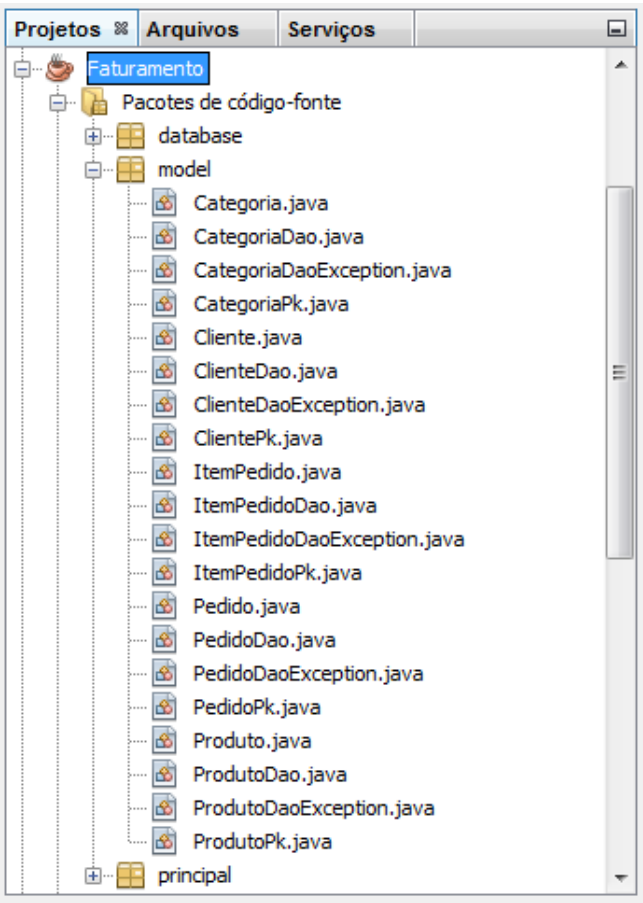

Figura 5.12 – Projeto com as classes geradas para a camada DAO
Fonte: Do próprio autor

5.3 XML e XMI ArgoUML

Esta seção apresenta o funcionamento da ferramenta de geração de código tendo como origem um documento XML.

XML (*Extensible Markup Language*) é um padrão de fato para estruturar dados a partir de uma linguagem de marcação com suas devidas especificações. XML foi projetado para armazenar e transportar dados (W3C, 2012). Este formato é muito popular, sendo utilizado em diversos tipos de aplicações Web, desktop e *mobile*.

Ferramentas de modelagem de dados UML frequentemente utilizam XML para armazenar a estrutura dos modelos desenhados, visto que em XML é natural a representação hierárquica de componentes, bem como a ideia de representar os vários atributos para cada elemento dos diagramas. Uma ferramenta que utiliza esse conceito é o ArgoUML. O arquivo de modelo de dados que este software armazena, com extensão **".zargo"**, nada mais é que um arquivo compactado que guarda em XML a estrutura dos diagramas UML que foram desenhados pelo programa. O ArgoUML também possui uma opção que permite exportar o modelo para o formato XMI (*XML Metadata Interchange)*, justamente para que possa ser integrado com outras ferramentas.

5.3.1 Formato da especificação

A especificação de entrada para XML foi escrita utilizando o componente JDOM. Este componente permite executar o *parser* de arquivos XML com bastante facilidade, desta forma foi escrito um algorítmo para transferir o conteúdo de um XML qualquer para a estrutura de dados hierárquica que é passada o núcleo do gerador de código. No caso de uso que trata este exemplo, de leitura da modelagem do ArgoUML, foi definida uma especialização da especificação de XML, identificando os elementos XMI que devem ser importados (no caso são usados os elementos *Class* e *Attribute*). A implementação segue a metodologia descrita por Fowler (2010) como uma das formas de tratar uma DSL (linguagem específica de domínio). Neste caso o domínio seria a descrição de um diagrama de classes, logo o *parser* é feito, conforme denominado pelo autor como uma "*Embedded Interpretation*".

Na Figura 5.13 pode ser visto o formato que uma classe de um determinado diagrama no arquivo XMI é armazenado pelo ArgoUML. Neste exemplo pode-se verificar trecho da classe “PessoaFisica”, (mostrada na linha 137) com os atributos “cpf” e “sexo”, (vistos nas linhas 144 e 159 respectivamente). Como o modelo para todas as classes fica bastante extenso, para fins de exemplo prático é mostrado apenas parte deste arquivo.

```
136   <UML:Class xmi.id = '10-2-5--33-49e308a9:115d9274332:-8000:00000000000007E2'
137     name = 'PessoaFisica' visibility = 'public' isSpecification = 'false' isRoot = 'false'
138     isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
139     <UML:GeneralizableElement.generalization>
140       <UML:Generalization xmi.idref = '10-2-5--33-415edcb7:115d94951c4:-8000:0000000000000827'/>
141     </UML:GeneralizableElement.generalization>
142     <UML:Classifier.feature>
143       <UML:Attribute xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:000000000000088E'
144         name = 'cpf' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
145         changeability = 'changeable' targetScope = 'instance'>
146         <UML:StructuralFeature.multiplicity>
147           <UML:Multiplicity xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:0000000000000894'>
148             <UML:Multiplicity.range>
149               <UML:MultiplicityRange xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:0000000000000893'
150                 lower = '1' upper = '1'/>
151             </UML:Multiplicity.range>
152           </UML:Multiplicity>
153         </UML:StructuralFeature.multiplicity>
154         <UML:StructuralFeature.type>
155           <UML:Class xmi.idref = '10-2-5--33--30c4ac56:1166948eac6:-8000:0000000000000897'/>
156         </UML:StructuralFeature.type>
157       </UML:Attribute>
158       <UML:Attribute xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:00000000000008A1'
159         name = 'sexo' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
160         changeability = 'changeable' targetScope = 'instance'>
161         <UML:StructuralFeature.multiplicity>
162           <UML:Multiplicity xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:0000000000000939'>
163             <UML:Multiplicity.range>
164               <UML:MultiplicityRange xmi.id = '10-2-5--33--30c4ac56:1166948eac6:-8000:0000000000000938'
165                 lower = '1' upper = '1'/>
166             </UML:Multiplicity.range>
167           </UML:Multiplicity>
168         </UML:StructuralFeature.multiplicity>
169         <UML:StructuralFeature.type>
170           <UML:DataType xmi.idref = '10-2-5--33--30c4ac56:1166948eac6:-8000:000000000000088D'/>
171         </UML:StructuralFeature.type>
172       </UML:Attribute>
```

Figura 5.13 – Formato das classes em XMI armazenado pelo ArgoUML

Fonte: Do próprio autor

5.3.2 Exemplo prático

Para a execução de um exemplo prático, foi utilizado um modelo de classes didático, de um sistema para uma corretora de seguros. A partir da especificação de entrada de XML é possível carregar informações sobre o modelo e gerar código fonte para as classes do sistema. A Figura 5.14 mostra o diagrama com as classes do sistema de corretora de seguros.

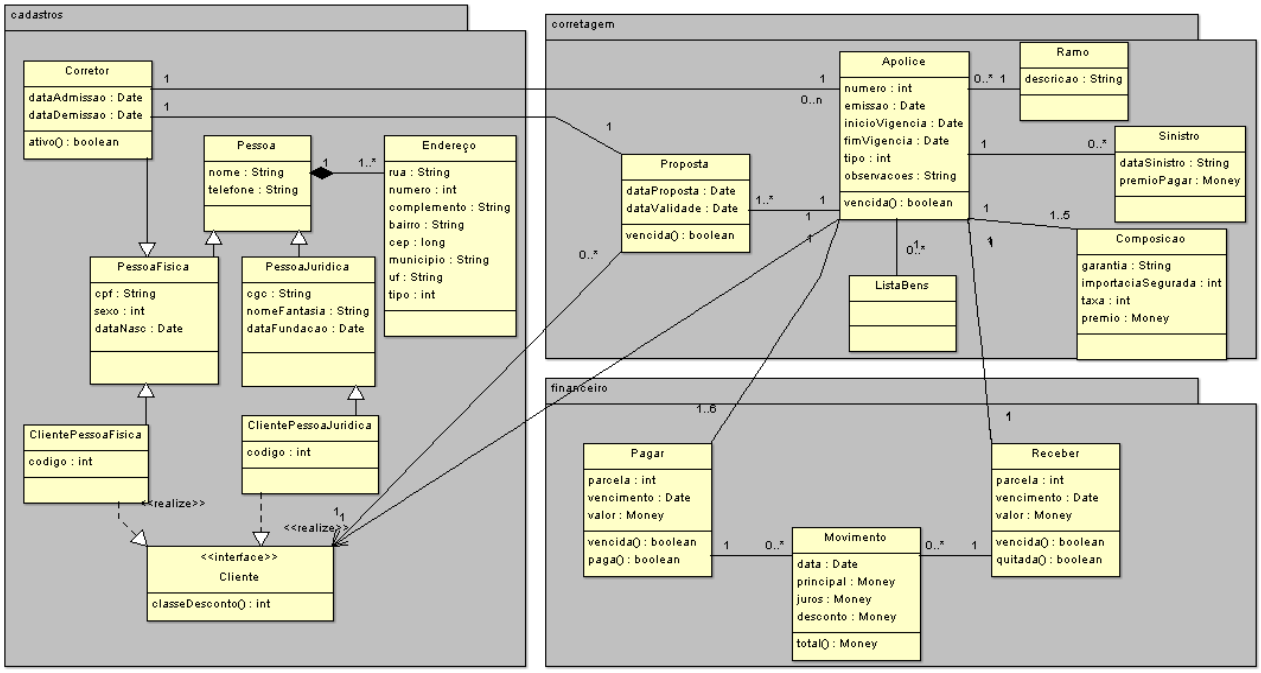

Figura 5.14 – Diagrama de classes de sistema para corretora de seguros
Fonte: Do próprio autor

5.3.3 Template

Foi utilizado o mesmo *template* que descreve a estrutura do código fonte a ser gerado para as classes POJO para o exemplo prático de geração da camada DAO a partir da estrutura de banco de dados. Esta é uma das premissas da solução proposta neste trabalho, sendo possível o reaproveitamento entre projetos distintos, utilizando-se os mesmos *templates* de outros projetos. Isto é possível devido a construção das especificações de entrada serem padronizadas, possuindo estruturas similares com os nomes das características dos elementos a serem substituídos sendo idênticos independente da origem dos dados.

5.3.4 Resultado da geração de código

O resultado da geração de código são classes Java geradas a partir de um modelo UML. Pode-se utilizar aqui a metodologia de geração de código passivo, ou seja, as classes geradas a partir do diagrama de classes do ArgoUML podem ser modificadas posteriormente

pelo desenvolvedor, servindo como base para a construção de uma aplicação maior. Há neste caso um aproveitamento da parte de análise mas não seria mais possível executar ciclos de desenho e geração do código, pois a abordagem de geração de código passivo permite que o desenvolvedor complemente o código gerado com código escrito manualmente. A Figura 5.15 mostra o projeto aberto na IDE Netbeans com as classes que foram geradas.

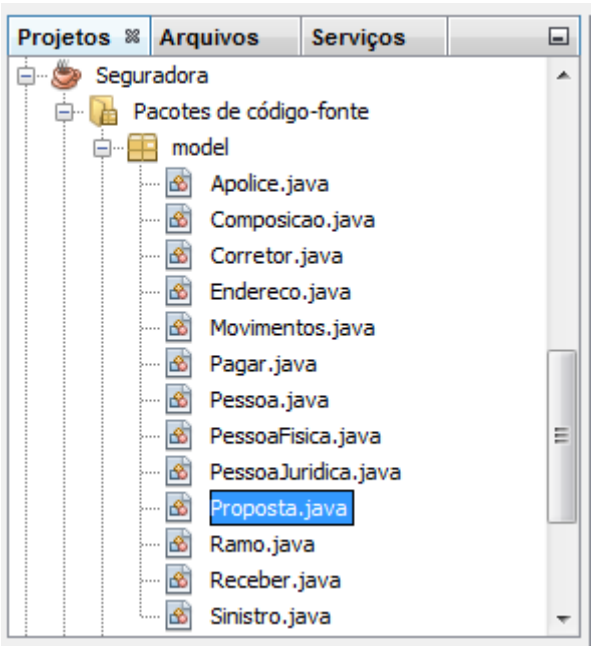

Figura 5.15 – Projeto com as classes geradas a partir de modelo XML
Fonte: Do próprio autor

As classes POJO geradas para cada entidade do modelo de dados encapsulam todos os atributos de cada objeto, a Figura 5.16 mostra parte da classe **Apolice**, que foi gerada a partir deste exemplo.

```
package model;
import java.util.Date;
/**
 * Classe que representa a entidade Apolice
 *
 * @author Sandro Silveira
 */
public class Apolice {
    /**
     * Identificador
     */
    public int id;
    /**
     * Número da apólice de seguro
     */
    public int numero;
    ...
    /**
     * Data de início de vigência
     */
    public Date iniciovigencia;
    ...
    /**
     * Construtor principal
     *
     */
    public Apolice() {
    }
    ...
    /**
     * Seta data de emissão
     */
    public void setEmissao(Date emissao) {
        this.emissao = emissao;
    }
```

Figura 5.16 – Trecho de código fonte gerado
Fonte: Do próprio autor

5.4 Dicionário de dados do SIGER

Nesta ultima seção que descreve os casos de usos da ferramenta de geração de código, será detalhado o funcionamento em conjunto com o ERP SIGER, gerando código para a camada de acesso a dados.

A Rech Informática LTDA, situada em Novo Hamburgo/RS, atua há mais de 20 anos como fornecedora de soluções em software integrado de gestão empresarial nas áreas de manufatura, estoques, compras, faturamento, financeiro, fiscal, contábil, recursos humanos, entre outros.

Utilizado por clientes de vários estados brasileiros em diversos segmentos industriais como metalúrgicas, químicas, calçadistas além de escritórios contábeis e comércios em geral, o SIGER é um ERP que oferece um amplo conjunto de ferramentas operacionais e gerenciais com uma interface padronizada e amigável ao usuário. Estima-se que este software seja utilizado diariamente por mais de 6000 pessoas. O uso do SIGER permite a seus usuários atenderem completamente a complexa legislação brasileira e oferece muitos benefícios como a eliminação de retrabalho e consolidação de informações para a tomada de decisões.

O SIGER é um software totalmente parametrizável e com muitas customizações. É desenvolvido em COBOL, Java e Delphi, com bases de dados ISAM, Ctree e Oracle, utiliza também as ferramentas SP2 e Formprint da americana Flexus. O software ao todo, possui mais de 8.300.000 linhas de código, cerca de 4.300 programas, 1.200 tabelas na base de dados, 3.500 janelas de interface gráfica com o usuário (GUI) e mais de 1.000 formulários gráficos para impressão. Tudo isto é desenvolvido e mantido por uma equipe com mais de 40 analistas e programadores que estudam constantemente formas de melhorar o produto e atender com maior flexibilidade e qualidade à crescente demanda apresentada pelo mercado. O suporte técnico é composto por 50 profissionais com forte capacitação contábil, fiscal e financeira, que mantém o SIGER em funcionamento em mais de 700 empresas.

Um dos pilares para a evolução constante do SIGER é um dicionário de dados que mantém todas as informações sobre a estrutura da base de dados do ERP. O dicionário de dados é uma ferramenta importante, pois é a partir dele que diversas aplicações buscam informações para atender de forma dinâmica à necessidades tais como: elaboração de relatórios personalizados, seleção padrão, leitura padrão, validação de dados, extração de dados, geração de código fonte e integração com rotinas da interface com o usuário. O dicionário de dados do SIGER mantém informações sobre todas as tabelas, campos

(atributos), chaves, valores enumerados e relacionamentos do ERP. O dicionário é uma ferramenta interativa que é alimentada pelos desenvolvedores com as estruturas de dados e também acompanha a instalação do SIGER, visto que fornece funcionalidades em tempo de desenvolvimento e de execução. A Figura 5.17, apresenta um diagrama resumido com os principais elementos do dicionário de dados do SIGER.

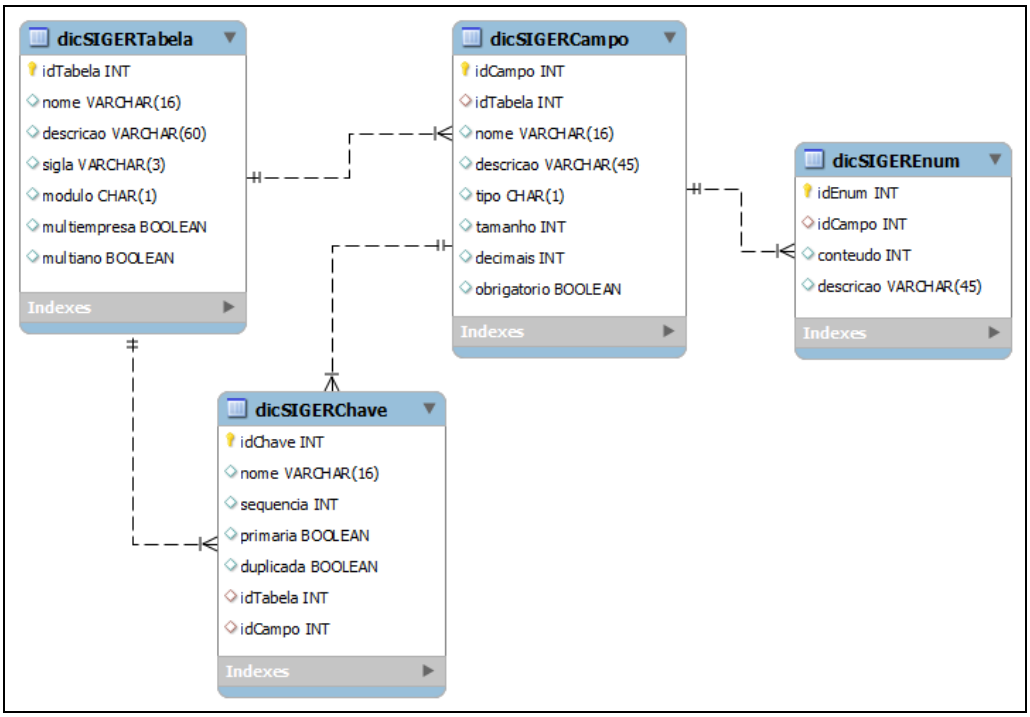

Figura 5.17 – Diagrama resumido do dicionário de dados do SIGER
Fonte: Do próprio autor

5.4.1 Formato da especificação

A especificação de entrada desenvolvida para fornecer informações ao núcleo do gerador de código a partir do dicionário de dados busca informações sobre as tabelas do SIGER e as disponibiliza para geração de código.

A classe de especificação do dicionário de dados do SIGER faz a leitura das tabelas que foram apresentadas de forma resumida na Figura 5.17, e coloca os dados em uma estrutura similar a utilizada pela especificação de entrada para banco de dados. Desta forma os mesmos *templates* criados anteriormente podem ser utilizados, demonstrando que é possível modificar-se a origem dos dados e que o processamento de saída será consistente com o projeto da ferramenta de geração de código, que prevê que várias especificações diferentes podem gerar o mesmo tipo de artefato.

5.4.2 Exemplo prático

Para facilitar a configuração de um conjunto de classes a serem geradas a partir do dicionário de dados, foi utilizada a interface gráfica com o usuário que foi desenvolvida na forma de *wizard* (assistente). O objetivo é solicitar passo a passo cada elemento necessário para a geração de cada classe.

O assistente de geração de código é composto por várias etapas em que são solicitadas as opções da ferramenta, que são: especificação de entrada, elemento de origem, *template* e arquivo de saída. As figuras a seguir exemplificam cada uma das etapas para a criação de uma classe POJO a partir da tabela de Plano Contábil do SIGER.

Na Figura 5.18 verifica-se que é possível selecionar qual especificação de entrada será utilizada, sendo que para as especificações de banco de dados e dicionário de dados do SIGER são exibidos campos relativos à conexão com o banco de dados via JDBC.

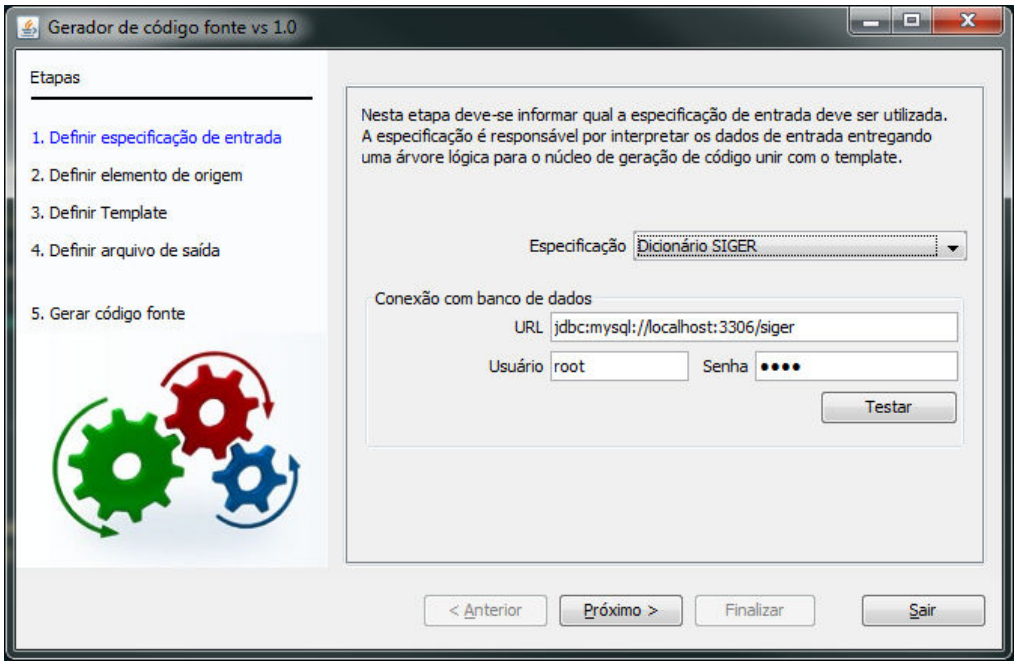

Figura 5.18 – Wizard/Especificação de entrada
Fonte: Do próprio autor

A Figura 5.19 mostra a etapa que permite informar qual a tabela deve ser utilizada como elemento base para a geração de código.

Figura 5.19 – Wizard/Definição de elemento de origem
Fonte: Do próprio autor

A Figura 5.20 mostra a etapa que permite informar o *template* utilizado para geração de código, ao pressionar o botão "..." é exibida uma caixa de diálogo de seleção de arquivos.

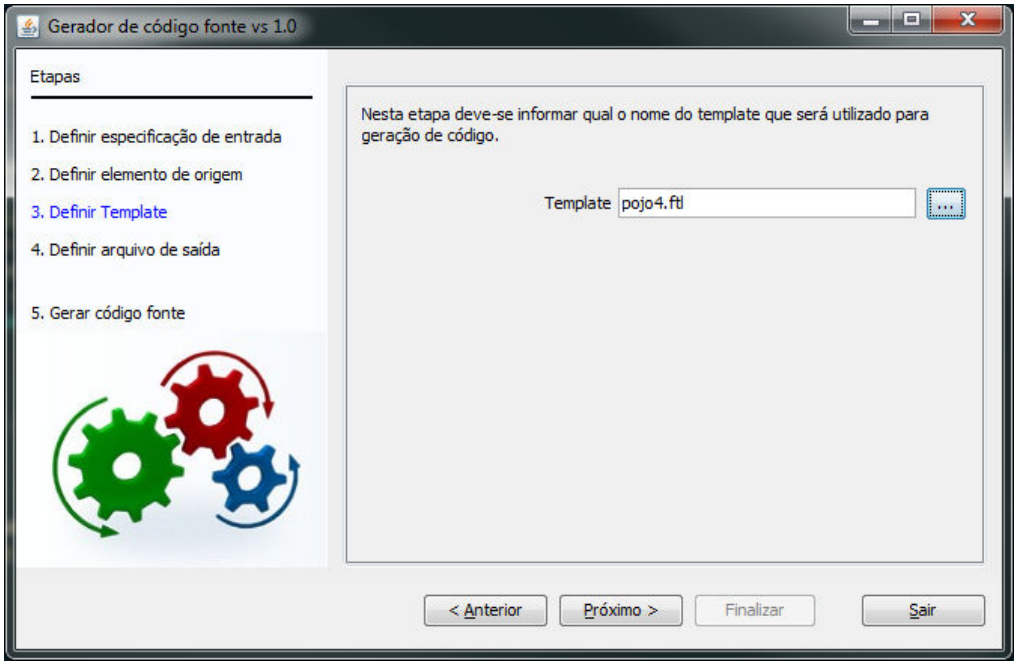

Figura 5.20 – Wizard/Definição de *Template*
Fonte: Do próprio autor

A Figura 5.21 mostra a etapa que permite informar o caminho e nome do arquivo que será utilizado para gravar a saída do código fonte gerado.

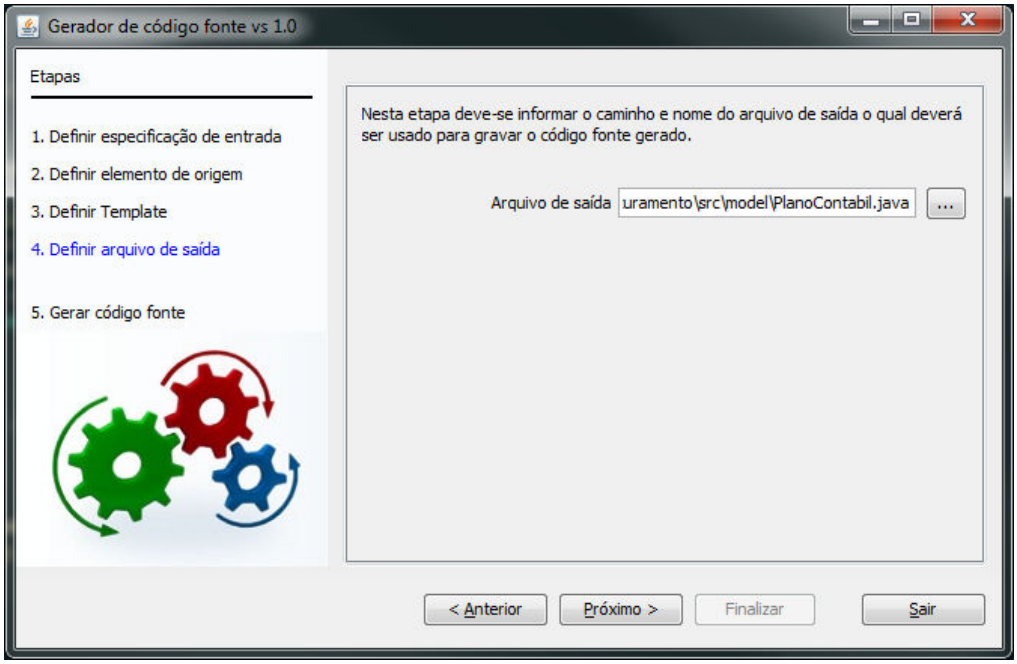

Figura 5.21 – Wizard/Definição do arquivo de saída
Fonte: Do próprio autor

A Figura 5.22 mostra a última etapa, com o botão "Finalizar" habilitado para gerar o código fonte. Também permite a criação de um novo projeto ou adicionar a configuração criada à um projeto existente para uso posterior.

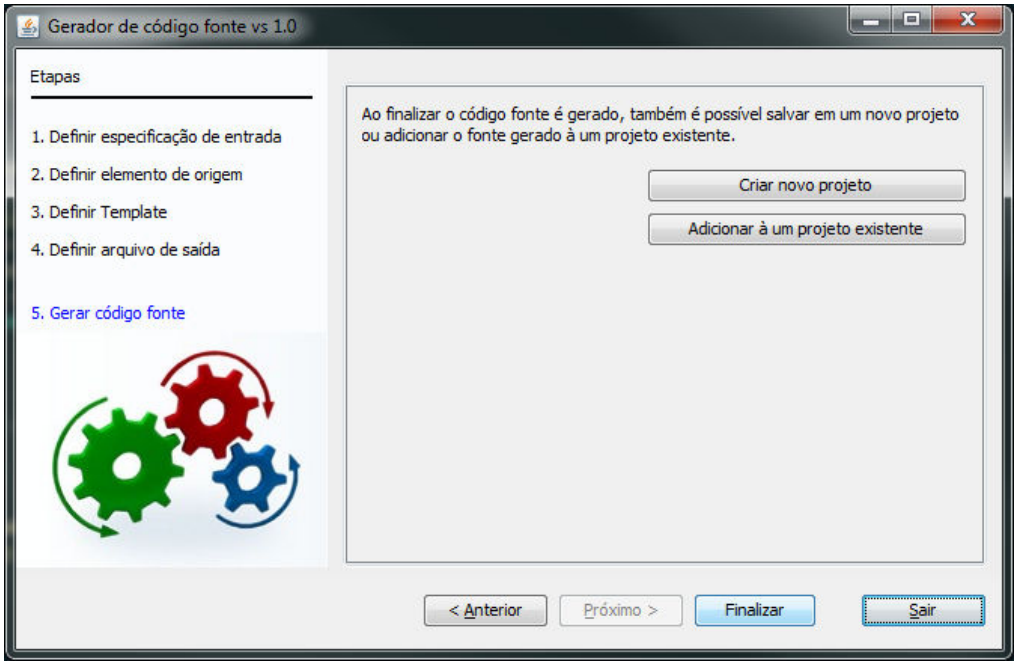

Figura 5.22 – Wizard/Geração do código fonte
Fonte: Do próprio autor

5.4.3 Templates

Para validação da ferramenta de geração de código, foram geradas as classes de acesso a dados para algumas tabelas do ERP SIGER da Rech Informática LTDA. A escolha de gerar estas classes a partir de uma nova especificação de entrada, que monte as estruturas de dados a partir do dicionário de dados do SIGER e não diretamente dos metadados do banco de dados oferece muitas vantagens. Com o objetivo de traduzir de forma mais consistente a estrutura que o sistema organiza e armazena estes dados, esta abordagem permite especificar melhor os detalhes da implementação e gera uma documentação mais ampla.

Além das vantagens citadas, o uso do dicionário de dados permite a integração com rotinas de validação de regras de negócio do sistema e detalhamento preciso de campos multivalorados (denominados no SIGER como campos enumerados), para o campo **sexo**, por exemplo, o dicionário de dados descreve que M=Masculino e F=Feminino. Para esta necessidade pontual do projeto em que será gerado código para este exemplo prático foi criado um novo *template*, denominado **pojo6.ftl**, o qual faz a iteração em mais um nível, ou seja, dentro de um atributo, é possível varrer seus valores possíveis, permitindo gerar métodos específicos para atribuir e retornar verdadeiro ou falso para cada um deles. Seguindo o mesmo exemplo do campo **sexo**, é possível então criar os métodos **setMasculino, setFeminino, isMasculino** e **isFeminino**, facilitando o uso dessas classes, visto que parte das regras de negócio precisariam descrever isto, e a geração de código aqui mais uma vez economiza esse tempo importante e também mantém a consistência dos dados na camada da aplicação.

5.4.4 Resultado da geração de código

As classes geradas a partir do dicionário de dados do SIGER serão utilizadas para o desenvolvimento de um módulo Web que está sendo desenvolvido. Esta metodologia, de trabalhar com especificações separadas do núcleo de geração de código, o qual utiliza um componente próprio e com muitos recursos disponíveis para manipular dados nos *templates,* garantiu uma grande flexibilidade, que não era possível com a ferramenta utilizada pela Rech Informática. A rapidez de montar um *template*, visualizar os dados de entrada e refinar o resultado final garante que um grande número de classes de acesso a dados possam ser geradas com um ganho de produtividade expressivo. O tempo de montar um *template* a partir de um modelo e colocá-lo totalmente funcional fica na casa de minutos, por exemplo, o *template* **Dao1.java** foi desenvolvido a partir de um modelo pré-existente de DAO e não levou mais que 15 minutos para ser refinado, gerando classes completamente funcionais a

partir de sua utilização. No formato tradicional, utilizado anteriormente, seria necessário implementar alguns scripts de geração de código direto no núcleo da ferramenta de dicionário do SIGER, o qual já é bastante complexo para atender as necessidades do ERP, para permitir atender algumas necessidades específicas deste modelo DAO, esta implementação levaria em torno de 8 horas para ser desenvolvido. Estas estimativas foram feitas levando em consideração implementações anteriores para criar novos *scripts* ou acrescentando lógicas que no modelo proposto por este trabalho podem ser resolvidos com recursos da linguagem de *template* do Freemarker.

5.5 Integração com outras ferramentas e trabalhos futuros

O gerador de código fonte desenvolvido neste trabalho a partir de um *framework* expansível, permite uma grande flexibilidade combinando entradas existentes com *templates* escritos com o FreeMarker. Novas entradas são facilmente implementadas, bastando escrever uma nova classe que utilize uma classe base e uma interface padrão que obriga que alguns métodos sigam um modelo pré-definido. Algumas especificações de dados novas podem até mesmo seguir técnicas de criação de compiladores, criando-se módulos de tratamento léxico e sintático que carregue as especificações em uma árvore sintática, que pode ser usada posteriormente para junto aos *templates* transformar a ferramenta de geração de código em um poderoso tradutor de linguagens. Algumas especificações que são possíveis de implementar:

- Especificação de entrada a partir de arquivos *properties*;
- Especificação para leitura de formulários HTML;
- Especificação para leitura de formulários DFM Delphi;
- Estruturas de dados oriundas de qualquer linguagem de programação ou DSL.

É possível não só expandir o uso da ferramenta facilmente como também integrá-la a outras ferramentas de desenvolvimento. Neste trabalho não foi criado um *plugin* para que a ferramenta funcione integrada as IDEs Java como NetBeans e Eclipse, visto que esse desenvolvimento tomaria um tempo importante que foi utilizado para expandir mais alguns exemplos práticos que comprovassem a flexibilidade e expansibilidade e experimentação criando várias classes para cada exemplo proposto, com diferentes tipos de dados e aplicações. A criação de uma interface no modo *Wizard*, a fim de facilitar a entrada de cada uma das opções de execução sendo salvas ao final em um arquivo de projeto também

suplementa a falta da integração da ferramenta de geração de código diretamente em uma IDE conhecida. Desta forma seu uso fica também independente de ferramenta de desenvolvimento e até mesmo linguagem de programação, visto que desenvolvedores que utilizem Delphi, PHP, Visual Basic, C#, etc, também podem utilizá-la em seus projetos.

CONCLUSÃO

As vantagens em utilizar ferramentas de geração de código são várias, dentre as quais destacam-se o ganho de qualidade e produtividade quando estas ferramentas são bem utilizadas. A criação de uma nova ferramenta de geração de código por vezes é necessária para atender necessidades específicas. Ao iniciar o projeto de uma nova ferramenta de geração de código, defronta-se com uma questão: desenvolver toda a ferramenta sem aproveitar nenhum código já desenvolvido ou utilizar-se de padrões de projeto, frameworks e *templates engines* existentes.

O estudo de padrões de projeto e *frameworks* confirmou a importância destes conceitos na criação de uma ferramenta que seja flexível, reutilizável e eficiente para geração de código. A facilidade de incluir novas funcionalidades é importante em qualquer software. Se o mesmo for construído a partir de padrões conhecidos, facilita a evolução e minimiza o número de erros.

O estudo sobre DSL traz uma visão importante sobre seu escopo limitado, e permite compreender como deve ser a atuação das linguagens de *template*, que não devem ter a pretensão de substituir o que se faz com uma linguagem de programação de propósito geral. Por sua vez, uso de *templates engines* permite adicionar muitos recursos úteis na geração de código.

Ainda a partir dos estudos realizados, foi possível concluir que cada vez mais ferramentas de modelagem e metodologias convergem no sentido de que a expressão visual é importante para a definição de software. O uso de UML comprova isto. A modelagem permite que as ferramentas de desenvolvimento façam uso dos modelos criados com a finalidade de gerar código fonte de forma automatizada, ao invés dos desenvolvedores terem de escrevê-lo.

As conclusões acima foram importantes para embasar a decisão de criar a ferramenta utilizando padrões de projeto e *templates engines* existentes. Esta escolha mostrou-se acertada, visto que o resultado final foi uma ferramenta flexível e expansível, criada a partir do *framework* que é objeto deste trabalho.

A ferramenta foi concebida com uma estrutura modular. A utilização de componentes especializados deixou a ferramenta preparada para expansão futura. O resultado no entendimento da arquitetura e organização da ferramenta foi bem maior do que se fosse desenvolvida de forma unificada e com um fim específico.

A execução de vários exemplos práticos serviu para validar a ferramenta, inclusive a partir da geração de código fonte baseado em dicionário de dados para o ERP SIGER. A conclusão final é a de que o uso do *framewok* apresentado neste trabalho é muito prático e permite a geração de código a partir de virtualmente qualquer origem de dados. A utilização tanto em linha de comando quanto via interface gráfica, garante que a ferramenta criada possa integrar-se a projetos de desenvolvimento de software nas mais variadas tecnologias e sistemas operacionais.

REFERÊNCIAS BIBLIOGRÁFICAS

ARNOLD, Ken; GOSLING, James; HOLMES, David. **A linguagem de Programação Java**. Porto Alegre: Bookman, 2007. 800p.

BECK, Kent. **Programação extrema Explicada: acolha as mudanças.** Porto Alegre: Bookman, 2004. 182 p.

BRAUDE, Eric. **Projeto de software:** da programação à arquitetura: uma abordagem baseada em Java**.** Porto Alegre: Bookman, 2005. 619 p.

CZARNECKI, Krysztof. **Generative Programming:** Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Dissertation for degree of Doktor-Ingenieur. Department of Computer Science and Automation Technical University of Ilmenau. Ilmenau – Alemanha. 1998. 449 p.

CZARNECKI, Krysztof; EISENECKER, Ulrich. **Generative Programming:** Methods, Tools, and Applications. Addison-Wesley Professional, 2000. 832 p.

ECLIPSE EMF: Eclipse Modeling Framework Project. Disponível em <http://eclipse.org/modeling/emf>. Acesso em 21 nov. 2011.

FOWLER, Martin; SCOTT Kendall. **UML essencial:** Um breve guia para a linguagem-padrão de modelagem de objetos. Porto Alegre: Bookman, 2005. 160 p.

FOWLER, Martin. **Domain specific languages.** Addison-Wesley Professional, 2010. 597 p.

FREEMARKER: **Java Template Engine Library**. Disponível em <http://freemarker.sourceforge.net/docs/preface.html>. Acesso em 15 set. 2011.

GAMMA, Erich et al. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. 364 p.

GOODRICH, Michael T.; TOMASSIA, Robert. **Estruturas de Dados e Algoritmos em Java**. 2. ed. Porto Alegre: Bookman, 2002. 584 p.

GRADECKI, Joseph D.; COLE, Jim. **Mastering Apache Velocity**: Java Open Source Library. Wiley. Indianapolis, 2003. 375 p.

HERRINGTON, Jack. **Code generation in action.** Greenwich: Manning, 2003. 372 p.

KREISIG, Mariana. **PROPOSTA DE AMBIENTE DE COOPERAÇÃO PARA DESENVOLVIMENTO DE SOFTWARE INTEGRADO AO FRAMEWORK JFACE.** 2007. Trabalho de Conclusão de Curso. Feevale, Novo Hamburgo, RS.

LARMAN, Craig. **Utilizando UML e padrões**. Uma introdução à análise e ao projeto orientado a objetos e ao desenvolvimento interativo. Porto Alegre: Bookman, 2007. 696 p.

MCCONNELL, Steve. **Code complete:** um guia prático para a construção de software. Porto Alegre: Bookman, 2005. 928 p.

MELLOR, Stephen J. et al. **MDA Distilled:** Principles of Model-Driven Architecture. Addison-Wesley Professional, 2004. 176 p.

MySQL Documentation: **Java, JDBC and MySQL Types**. Disponível em <http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-type-conversions.html> Acesso em 24 mai. 2012.

OMG: **Object Management Group**. Disponível em <http://www.omg.org>. Acesso em: 20 out. 2011.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron, 1995. 1056 p.

PRESSMAN, Roger S. **Engenharia de Software**. 6. ed. Porto Alegre: AMGH, 2010. 720 p.

SEBESTA, Robert W. **Conceitos de linguagens de programação.** 4. ed. Porto Alegre, RS: Bookman, 2000. 624 p.

VELOCITY: **The Apache Velocity Project.** Disponível em <http://velocity.apache.org>. Acesso em 15 set. 2011.

W3C: **XML**: Extensible Markup Language**.** Disponível em < http://www.w3.org/XML/>. Acesso em 06 jun. 2012.

WALLS, Craig; RICHARDS, Norman. **XDoclet in Action**. Greenwich, UK: Manning, 2004. 591 p.

APÊNDICE A – CÓDIGO GERADO A PARTIR DE ARQUIVO INI

Livro.java

```java
/*
 * Autor: Sandro Silveira
 *
 */
package model;

public class Livro {

    /**
     * Título do livro
     */
    private String titulo;

    /**
     * Autor do livro
     */
    private Autor autor;

    /**
     * Editora do livro
     */
    private Editora editora;

    /**
     * Local de edição
     */
    private String local;

    /**
     * Ano de publicação
     */
    public int ano;

    /**
     * Número de páginas
     */
    private int paginas;

    /**
     * Se está disponível para empréstimo
     */
    private boolean disponivel;

    /**
     * Seta título do livro
     */
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    /**
     * Retorna título do livro
     */
    public String getTitulo() {
        return titulo;
```

```
56      }
57
58      /**
59       * Seta autor do livro
60       */
61      public void setAutor(Autor autor) {
62          this.autor = autor;
63      }
64
65      /**
66       * Retorna autor do livro
67       */
68      public Autor getAutor() {
69          return autor;
70      }
71
72      /**
73       * Seta editora do livro
74       */
75      public void setEditora(Editora editora) {
76          this.editora = editora;
77      }
78
79      /**
80       * Retorna editora do livro
81       */
82      public Editora getEditora() {
83          return editora;
84      }
85
86      /**
87       * Seta local de edição
88       */
89      public void setLocal(String local) {
90          this.local = local;
91      }
92
93      /**
94       * Retorna local de edição
95       */
96      public String getLocal() {
97          return local;
98      }
99
100     /**
101      * Seta ano de publicação
102      */
103     public void setAno(int ano) {
104         this.ano = ano;
105     }
106
107     /**
108      * Retorna ano de publicação
109      */
110     public int getAno() {
111         return ano;
112     }
113
114     /**
```

```
115      * Seta número de páginas
116      */
117     public void setPaginas(int paginas) {
118         this.paginas = paginas;
119     }
120 
121     /**
122      * Retorna número de páginas
123      */
124     public int getPaginas() {
125         return paginas;
126     }
127 
128     /**
129      * Seta se está disponível para empréstimo
130      */
131     public void setDisponivel(boolean disponivel) {
132         this.disponivel = disponivel;
133     }
134 
135     /**
136      * Retorna verdadeiro se está disponível para empréstimo
137      */
138     public boolean isDisponivel() {
139         return disponivel;
140     }
141 
142 }
143 
```

APÊNDICE B – CÓDIGO GERADO A PARTIR DE BANCO DE DADOS

Produto.java

```
 1 /*
 2  *  Copyright (C) 2012 Sandro Silveira
 3  *  sandrosilveira@gmail.com
 4  *
 5  *  Este programa foi desenvolvido para o trabalho de conclusão de
 6  *  curso apresentado como requisito parcial à obtenção do grau de
 7  *  Bacharel em Ciência da Computação pela Universidade Feevale.
 8  *
 9  */
10 package model;
11
12 public class Produto {
13
14 /**
15  * Classe que representa a entidade Produto
16  *
17  * @author Sandro Silveira
18  */
19     /**
20      * Identificador
21      */
22     private int id;
23
24     /**
25      * Nome completo
26      */
27     private String nome;
28
29     /**
30      * Preço de venda
31      */
32     private java.math.BigDecimal preco;
33
34     /**
35      * Descrição detalhada
36      */
37     private String descricao;
38
39     /**
40      * Data da última alteração
41      */
42     private java.sql.Date dataAlteracao;
43
44     /**
45      * Categoria
46      */
47     private int idCategoria;
48
49     /**
50      * Construtor principal
51      *
52      */
53     public Produto() {
54     }
55
```

```
56      /**
57       * Seta o Identificador
58       */
59      public void setId(int id) {
60          this.id = id;
61      }
62
63      /**
64       * Retorna o Identificador
65       */
66      public int getId() {
67          return id;
68      }
69
70      /**
71       * Seta Nome completo
72       */
73      public void setNome(String nome) {
74          this.nome = nome;
75      }
76
77      /**
78       * Retorna Nome completo
79       */
80      public String getNome() {
81          return nome;
82      }
83
84      /**
85       * Seta Preço de venda
86       */
87      public void setPreco(java.math.BigDecimal preco) {
88          this.preco = preco;
89      }
90
91      /**
92       * Retorna Preço de venda
93       */
94      public java.math.BigDecimal getPreco() {
95          return preco;
96      }
97
98      /**
99       * Seta Descrição detalhada
100      */
101     public void setDescricao(String descricao) {
102         this.descricao = descricao;
103     }
104
105     /**
106      * Retorna Descrição detalhada
107      */
108     public String getDescricao() {
109         return descricao;
110     }
111
112     /**
113      * Seta Data da última alteração
114      */
```

```
115     public void setDataAlteracao(java.sql.Date dataAlteracao) {
116         this.dataAlteracao = dataAlteracao;
117     }
118 
119     /**
120      * Retorna Data da última alteração
121      */
122     public java.sql.Date getDataAlteracao() {
123         return dataAlteracao;
124     }
125 
126     /**
127      * Seta Categoria
128      */
129     public void setIdCategoria(int idCategoria) {
130         this.idCategoria = idCategoria;
131     }
132 
133     /**
134      * Retorna Categoria
135      */
136     public int getIdCategoria() {
137         return idCategoria;
138     }
139 
140     @Override
141     public String toString() {
142         return "Produto{" +
143                     "id=" + id + ", " +
144                     "nome=" + nome + ", " +
145                     "preco=" + preco + ", " +
146                     "descricao=" + descricao + ", " +
147                     "dataAlteracao=" + dataAlteracao + ", " +
148                     "idCategoria=" + idCategoria + "}";
149     }
150 
151     @Override
152     public boolean equals(Object obj) {
153         if (obj == null) {
154             return false;
155         }
156         if (obj == this) {
157             return true;
158         }
159         if (getClass() != obj.getClass()) {
160             return false;
161         }
162         final Produto outroProduto = (Produto) obj;
163         if (this.id != outroProduto.id) {
164             return false;
165         }
166         if (this.nome != outroProduto.nome) {
167             return false;
168         }
169         if (this.preco != outroProduto.preco) {
170             return false;
171         }
172         if (this.descricao != outroProduto.descricao) {
173             return false;
```

```
174         }
175         if (this.dataAlteracao != outroProduto.dataAlteracao) {
176             return false;
177         }
178         if (this.idCategoria != outroProduto.idCategoria) {
179             return false;
180         }
181         return true;
182     }
183 
184 }
185 
```

ProdutoDao.java

```
  1 /*
  2  *  Copyright (C) 2012 Sandro Silveira
  3  *  sandrosilveira@gmail.com
  4  *
  5  *  Este programa foi desenvolvido para o trabalho de conclusão de
  6  *  curso apresentado como requisito parcial à obtenção do grau de
  7  *  Bacharel em Ciência da Computação pela Universidade Feevale.
  8  *
  9  */
 10 package model;
 11 
 12 import database.DBStaticConn;
 13 import model.Produto;
 14 import model.ProdutoPk;
 15 import model.ProdutoDaoException;
 16 import java.sql.Connection;
 17 import java.sql.PreparedStatement;
 18 import java.sql.ResultSet;
 19 import java.sql.SQLException;
 20 import java.util.ArrayList;
 21 import java.util.Collection;
 22 import java.util.Date;
 23 
 24 /**
 25  * Classe DAO (Data Access Object) para acesso a tabela PRODUTO
 26  *
 27  * @author Sandro Silveira
 28  */
 29 public class ProdutoDao {
 30 
 31     /**
 32      * Conexão com o banco de dados
 33      */
 34     private Connection conexao;
 35     /**
 36      * Métodos de busca podem passar este valor para o método JDBC setMaxRows
 37      */
 38     private int maxLinhas;
 39     /**
 40      * Indica se deve registrar trace de operações na tabela PRODUTO
 41      */
 42     private boolean registraTrace = false;
 43     /**
 44      * Comando Select padrão para métodos de busca da tabela PRODUTO
```

```
45      */
46     private final String SQL_SELECT = "SELECT id, nome, preco, descricao,
data_alteracao, id_categoria FROM PRODUTO";
47     /**
48      * Comando Insert padrão para a tabela PRODUTO
49      */
50     protected final String SQL_INSERT = "INSERT INTO PRODUTO (id, nome, preco,
descricao, data_alteracao, id_categoria) VALUES (?, ?, ?, ?, ?, ?)";
51     /**
52      * Comando Update padrão para a tabela PRODUTO
53      */
54     protected final String SQL_UPDATE = "UPDATE PRODUTO SET id = ?, nome = ?,
preco = ?, descricao = ?, data_alteracao = ?, id_categoria = ? WHERE id = ?";
55     /**
56      * Comando Delete padrão para a tabela PRODUTO
57      */
58     protected final String SQL_DELETE = "DELETE FROM PRODUTO WHERE id = ?";
59     /**
60      * Comando Delete para exclusão de todos os dados da tabela PRODUTO
61      */
62     protected final String SQL_DELETE_ALL = "DELETE FROM PRODUTO";
63     /**
64      * Indice da coluna id
65      */
66     protected static final int COLUNA_ID = 1;
67     /**
68      * Indice da coluna nome
69      */
70     protected static final int COLUNA_NOME = 2;
71     /**
72      * Indice da coluna preco
73      */
74     protected static final int COLUNA_PRECO = 3;
75     /**
76      * Indice da coluna descricao
77      */
78     protected static final int COLUNA_DESCRICAO = 4;
79     /**
80      * Indice da coluna data_alteracao
81      */
82     protected static final int COLUNA_DATA_ALTERACAO = 5;
83     /**
84      * Indice da coluna id_categoria
85      */
86     protected static final int COLUNA_ID_CATEGORIA = 6;
87     /**
88      * Número de colunas da tabela
89      */
90     protected static final int NRO_COLUNAS_TABELA = 6;
91
92     /**
93      * Construtor padrão
94      */
95     public ProdutoDao() {
96     }
97
98     /**
99      * Construtor informando a conexão com o banco de dados
100     * @param conn Conexão com o banco de dados
```

```
101      */
102     public ProdutoDao(Connection conexaoUsuario) {
103         this.conexao = conexaoUsuario;
104     }
105 
106     /**
107      * Seta o número máximo de colunas a retornar
108      */
109     public void setMaxRows(int maxRows) {
110         this.maxLinhas = maxRows;
111     }
112 
113     /**
114      * Retorna o número máximo de linhas a retornar
115      */
116     public int getMaxRows() {
117         return maxLinhas;
118     }
119 
120     /**
121      * Indica que deve registrar trace de operações na tabela PRODUTO
122      */
123     public void setTraceOn() {
124         this.registraTrace = true;
125     }
126 
127     /**
128      * Indica que não deve registrar trace de operações na tabela PRODUTO
(default)
129      */
130     public void setTraceOff() {
131         this.registraTrace = false;
132     }
133 
134     /**
135      * Registra trace de operações na tabela PRODUTO
136      * @param Texto a registrar, mostrando o passo-a-passo de execução
137      */
138     private void trace(String texto) {
139         if (registraTrace) {
140             System.out.println(texto);
141         }
142     }
143 
144     /**
145      * Insere uma nova linha na tabela PRODUTO
146      *
147      * @param produto Referência para o objeto a inserir
148      * @return Primary Key
149      * @throws ProdutoDaoException
150      */
151     public ProdutoPk insert(Produto produto) throws ProdutoDaoException {
152         long t1 = System.currentTimeMillis();
153         final boolean isConnSupplied = (conexao != null);
154         Connection conn = null;
155         PreparedStatement stmt = null;
156         try {
157             conn = isConnSupplied ? this.conexao :
DBStaticConn.getConnection();
```

```
158             stmt = conn.prepareStatement(SQL_INSERT);
159             stmt.setInt(COLUNA_ID, produto.getId());
160             stmt.setString(COLUNA_NOME, produto.getNome());
161             stmt.setBigDecimal(COLUNA_PRECO, produto.getPreco());
162             stmt.setString(COLUNA_DESCRICAO, produto.getDescricao());
163             stmt.setDate(COLUNA_DATA_ALTERACAO, produto.getDataAlteracao());
164             stmt.setInt(COLUNA_ID_CATEGORIA, produto.getIdCategoria());
165             trace("Executando " + SQL_INSERT + " com: " + produto.toString());
166             int rows = stmt.executeUpdate();
167             long t2 = System.currentTimeMillis();
168             trace(rows + " linha incluída em (" + (t2 - t1) + " ms)");
169             return new ProdutoPk(produto.getId());
170         } catch (Exception ex) {
171             ex.printStackTrace();
172             throw new ProdutoDaoException("Falha ao inserir: " +
ex.getMessage(), ex);
173         } finally {
174             DBStaticConn.close(stmt);
175             if (!isConnSupplied) {
176                 DBStaticConn.close(conn);
177             }
178         }
179     }
180
181     /**
182      * Atualiza uma linha na tabela PRODUTO
183      * @param pk Primary key
184      * @param produto Referência para o objeto a atualizar
185      * @throws ProdutoDaoException
186      */
187     public void update(ProdutoPk pk, Produto produto) throws
ProdutoDaoException {
188         long t1 = System.currentTimeMillis();
189         final boolean isConnSupplied = (conexao != null);
190         Connection conn = null;
191         PreparedStatement stmt = null;
192         try {
193             conn = isConnSupplied ? this.conexao :
DBStaticConn.getConnection();
194             trace("Executando " + SQL_UPDATE + " com: " + produto.toString());
195             stmt = conn.prepareStatement(SQL_UPDATE);
196             stmt.setInt(COLUNA_ID, produto.getId());
197             stmt.setString(COLUNA_NOME, produto.getNome());
198             stmt.setBigDecimal(COLUNA_PRECO, produto.getPreco());
199             stmt.setString(COLUNA_DESCRICAO, produto.getDescricao());
200             stmt.setDate(COLUNA_DATA_ALTERACAO, produto.getDataAlteracao());
201             stmt.setInt(COLUNA_ID_CATEGORIA, produto.getIdCategoria());
202             int indexKey = NRO_COLUNAS_TABELA;
203             stmt.setInt(++indexKey, pk.getId());
204             int rows = stmt.executeUpdate();
205             long t2 = System.currentTimeMillis();
206             trace(rows + " linha atualizada em (" + (t2 - t1) + " ms)");
207         } catch (Exception _e) {
208             _e.printStackTrace();
209             throw new ProdutoDaoException("Falha ao atualizar: " +
_e.getMessage(), _e);
210         } finally {
211             DBStaticConn.close(stmt);
212             if (!isConnSupplied) {
```

```
213                 DBStaticConn.close(conn);
214             }
215         }
216     }
217 
218     /**
219      * Deleta um linha da tabela PRODUTO
220      * @param pk Primary Key
221      * @throws ProdutoDaoException
222      */
223     public void delete(ProdutoPk pk) throws ProdutoDaoException {
224         long t1 = System.currentTimeMillis();
225         final boolean isConnSupplied = (conexao != null);
226         Connection conn = null;
227         PreparedStatement stmt = null;
228         try {
229             conn = isConnSupplied ? this.conexao :
DBStaticConn.getConnection();
230             trace("Executando " + SQL_DELETE + " com: " + pk.toString());
231             stmt = conn.prepareStatement(SQL_DELETE);
232             int indexKey = 0;
233             stmt.setInt(++indexKey, pk.getId());
234             int rows = stmt.executeUpdate();
235             long t2 = System.currentTimeMillis();
236             trace(rows + " linha excluída em (" + (t2 - t1) + " ms)");
237         } catch (Exception _e) {
238             _e.printStackTrace();
239             throw new ProdutoDaoException("Falha ao excluir: " +
_e.getMessage(), _e);
240         } finally {
241             DBStaticConn.close(stmt);
242             if (!isConnSupplied) {
243                 DBStaticConn.close(conn);
244             }
245         }
246     }
247 
248     /**
249      * Deleta todas as linhas da tabela PRODUTO
250      * @throws ProdutoDaoException
251      */
252     public void deleteAll() throws ProdutoDaoException {
253         long t1 = System.currentTimeMillis();
254         final boolean isConnSupplied = (conexao != null);
255         Connection conn = null;
256         PreparedStatement stmt = null;
257         try {
258             conn = isConnSupplied ? this.conexao :
DBStaticConn.getConnection();
259             trace("Executando " + SQL_DELETE_ALL);
260             stmt = conn.prepareStatement(SQL_DELETE_ALL);
261             int rows = stmt.executeUpdate();
262             long t2 = System.currentTimeMillis();
263             trace(rows + " linhas excluídas em (" + (t2 - t1) + " ms)");
264         } catch (Exception _e) {
265             _e.printStackTrace();
266             throw new ProdutoDaoException("Falha ao excluir todas as linhas da
tabela: " + _e.getMessage(), _e);
267         } finally {
```

```
268             DBStaticConn.close(stmt);
269             if (!isConnSupplied) {
270                 DBStaticConn.close(conn);
271             }
272         }
273     }
274
275     /**
276      * Retorna uma linha da tabela PRODUTO pela Primary Key
277      *
278      * @param pk Primary Key
279      * @return Objeto do tipo Produto
280      * @throws ProdutoDaoException
281      */
282     public Produto buscaPrimaryKey(ProdutoPk pk) throws ProdutoDaoException {
283         return buscaPrimaryKey(pk.getId());
284     }
285
286     /**
287      * Retorna uma linha da tabela PRODUTO comparando os atributos da Primary
Key
288      *
289      * @param id
290      * @return Array de objetos do tipo Produto
291      * @throws ProdutoDaoException
292      */
293     public Produto buscaPrimaryKey(int id) throws ProdutoDaoException {
294         Produto produtos[] = buscaByDynamicSelect(SQL_SELECT + " WHERE id =
?", new Object[]{new Integer(id)});
295         return produtos.length == 0 ? null : produtos[0];
296     }
297
298     /**
299      * Retorna linhas da tabela PRODUTO de acordo com a sentença SQL
300      * @param sql SQL a executar
301      * @param sqlParams Parâmetros
302      * @return Array de objetos do tipo Produto
303      * @throws ProdutoDaoException
304      */
305     public Produto[] buscaPorSelectDinamico(String sql, Object[] sqlParams)
throws ProdutoDaoException {
306         final boolean isConnSupplied = (conexao != null);
307         Connection conn = null;
308         PreparedStatement stmt = null;
309         ResultSet rs = null;
310         try {
311             conn = isConnSupplied ? this.conexao :
DBStaticConn.getConnection();
312             final String SQL = sql;
313             trace("Executando " + SQL);
314             stmt = conn.prepareStatement(SQL);
315             stmt.setMaxRows(maxLinhas);
316             // Passa os parâmetros de seleção
317             for (int i = 0; sqlParams != null && i < sqlParams.length; i++) {
318                 stmt.setObject(i + 1, sqlParams[i]);
319             }
320             // Executa a query
321             rs = stmt.executeQuery();
322             // Busca os resultados
```

```
323             Collection resultList = new ArrayList();
324             while (rs.next()) {
325                 Produto produto = new Produto();
326                 produto.setId(rs.getInt(COLUNA_ID));
327                 produto.setNome(rs.getString(COLUNA_NOME));
328                 produto.setPreco(rs.getBigDecimal(COLUNA_PRECO));
329                 produto.setDescricao(rs.getString(COLUNA_DESCRICAO));
330                 produto.setDataAlteracao(rs.getDate(COLUNA_DATA_ALTERACAO));
331                 produto.setIdCategoria(rs.getInt(COLUNA_ID_CATEGORIA));
332                 resultList.add(produto);
333             }
334             Produto retorno[] = new Produto[resultList.size()];
335             resultList.toArray(retorno);
336             return retorno;
337         } catch (Exception _e) {
338             _e.printStackTrace();
339             throw new ProdutoDaoException("Erro: " + _e.getMessage(), _e);
340         } finally {
341             DBStaticConn.close(rs);
342             DBStaticConn.close(stmt);
343             if (!isConnSupplied) {
344                 DBStaticConn.close(conn);
345             }
346         }
347     }
348 }
349
```