



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

RHAYLSON SILVA DO NASCIMENTO

**GENJPA: UMA FERRAMENTA DIRIGIDA POR MODELOS PARA GERAÇÃO DE
CÓDIGO JAVA/JPA**

TERESINA, PIAUÍ

2017

RHAYLSON SILVA DO NASCIMENTO

GENJPA: UMA FERRAMENTA DIRIGIDA POR MODELOS PARA GERAÇÃO DE
CÓDIGO JAVA/JPA

Projeto apresentado à Banca Examinadora como requisito para aprovação na disciplina de Trabalho de Conclusão de Curso II do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí.

Orientador: Prof. M^e. Fernando Castelo
Branco Gonçalves Santana

TERESINA, PIAUÍ

2017

FICHA CATALOGRÁFICA

Sistema de Bibliotecas

Gerada automaticamente com dados fornecidos pelo(a) autor(a)

N244g Nascimento, Rhaylson Silva do
GenJPA : Uma Ferramenta Dirigida por Modelos para Geração de Código Java/JPA /
Rhaylson Silva do Nascimento - 2017.
85 p. : il. color.

Trabalho de conclusão de curso (Tecnologia) - Instituto Federal de Educação,
Ciência e Tecnologia do Piauí, Campus Teresina Central, Tecnologia em Análise e
Desenvolvimento de Sistemas, 2017.

Orientador : Prof Me. Fernando Castelo Branco Gonçalves Santana.

1. UML. 2. Diagramas de Classes. 3. Engenharia à Frente. 4. Java. 5. Java
Persistence API. I. Título.

CDD 004



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA-CENTRAL
DIRETORIA DE ENSINO
DEPARTAMENTO DE INFORMAÇÃO, AMBIENTE, SAÚDE E PRODUÇÃO ALIMENTÍCIA

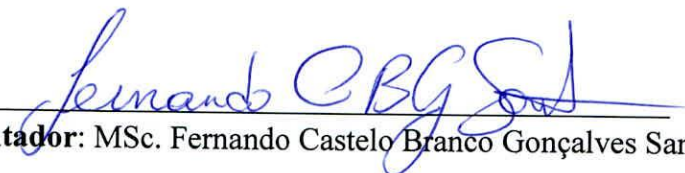
RHAYLSON SILVA DO NASCIMENTO

**GENJPA: UMA FERRAMENTA DIRIGIDA POR MODELOS PARA GERAÇÃO DE
CÓDIGO JAVA/JPA**

Monografia apresentada ao Curso de Tecnologia em
Análise e Desenvolvimento de Sistemas do Instituto
Federal de Ciência e Tecnologia do Piauí – IFPI, como
requisito para obtenção de grau de Tecnólogo em Análise
e Desenvolvimento de Sistemas.

Monografia aprovada em: 28 de junho de 2017.

Banca examinadora:


Orientador: MSc. Fernando Castelo Branco Gonçalves Santana


1ª Examinador: MSc. Duany Dreyton Bezerra Sousa


2ª Examinador: Dr. Franciéric Alves de Araújo

Dedico esta monografia à minha mãe Lúcia.

AGRADECIMENTOS

Agradeço, primeiramente, à Deus por ter me dado o dom da vida.

Aos meus familiares pelo suporte e pelos valores sociais a mim repassados.

Aos amigos que estiveram ao meu lado no decorrer desta jornada acadêmica, sempre muito solícitos e presentes.

Ao meu orientador Fernando Castelo Branco Gonçalves Santana pelo apoio e pela paciência despendida durante a elaboração desse trabalho.

Aos professores Duany Dreyton Bezerra Sousa e Fraciéric Alves de Araújo que aceitaram o convite para participar da banca de trabalho de conclusão de curso.

Aos mestres do curso de Tecnologia em Análise e Desenvolvimento de Sistemas por repassarem seus valiosos conhecimentos dessa área tão rica que é a computação.

Ao Instituto Federal de Educação Ciência e Tecnologia do Piauí por me fornecer a oportunidade de exercer pesquisas e estágios durante todo o curso.

RESUMO

A geração de código Java a partir de diagramas da UML, além de ser tema de estudo no meio acadêmico é prática recorrente no mercado de software. Diversas ferramentas, dentre elas as três maiores do mercado de modelagem implementam a técnica de engenharia à frente (transformação modelo-código) em Java. Entretanto, observa-se inconsistências no código gerado por muitas dessas aplicações. Conceitos mais complexos do diagrama de classes, como o de agregação, composição e navegabilidade deixam de ser transcritos durante a transformação, acarretando na quebra das restrições do projeto. Além disso, a interação com o usuário é uma função pouco explorada neste tipo de aplicação, limitando-se a apresentação de mensagens de erro. A constatação desses problemas serviu de motivação para elaboração da ferramenta apresentada nesta monografia, a *GenJPA*. Nela foi implementada uma engenharia à frente mais completa, baseada na linguagem Java e em anotações da *Java Persistence API*, capaz de mapear itens mais complexos do digrama de classes para o código e assim diminuir a ocorrência de inconsistências.

Palavras-chaves: UML, Diagramas de Classes, Engenharia à Frente, Java, Java Persistence API.

ABSTRACT

The generation of Java code from UML diagrams, besides a studied theme in academia, is a recurrent practice in the software market. Many tools, including the top three in modeling market, implement forward engineering techniques (model-code transformation) in Java. However, inconsistencies are observed in the generated code in many of these applications. More complex concepts of classes diagram, such as aggregation, composition and navigability are no longer transcribed during its transformation, resulting a breakdown on design constraints. In addition, user interaction is an under-exploited function on this application, limiting to a mere error message presentations. These problems were the motivation to the creation of the tool presented on these work, the *GenJPA*. It was implemented a more complete engineering, based on Java Language and notes from *Java Persistence API*, able to map more complex items from class diagram to the code, thus aiming to reduce the occurrence of inconsistencies.

Key-words: UML, Classes Diagram, Forward Engineering, Java, Java Persistence API.

LISTA DE ILUSTRAÇÕES

Figura 1 – Transformação Inconsistente de uma Composição [1].	16
Figura 2 – Transformação Inconsistente da Navegabilidade [1].	16
Figura 3 – Representação de Classes e Objetos.	20
Figura 4 – Organização dos Diagramas na UML. Adaptado da UML <i>Superstructure</i> [2].	21
Figura 5 – Diagrama de Classes que Apresenta Entidades. Adaptado de Booch e outros [3].	23
Figura 6 – <i>Model Driven Architecture</i> [4].	26
Figura 7 – Execução de um Programa Java. Adaptado de <i>Devmedia</i> [5].	28
Figura 8 – Classe Pessoa.	31
Figura 9 – Atributos Privados da Classe Pessoa.	31
Figura 10 – Herança na UML.	32
Figura 11 – Associação Binária Simples.	33
Figura 12 – Associação Unária ou Reflexiva.	33
Figura 13 – Exemplos de Agregação e Composição.	34
Figura 14 – Diagrama de Classes Modelado na <i>HiberObjects</i> . Adaptado de <i>Cgernert</i> [6].	36
Figura 15 – Diagrama de Classes no padrão <i>MD-JPA</i> [7]	37
Figura 16 – Interface da <i>GenJPA</i>	39
Figura 17 – Funcionamento da <i>GenJPA</i>	40
Figura 18 – Classe <code>CarregaArquivoXML.java</code>	43
Figura 19 – Classes de Modelo da <i>GenJPA</i>	44
Figura 20 – <code>Erro.java</code>	45
Figura 21 – <i>Feedback</i> ao Usuário.	47
Figura 22 – Arquivo de <i>Log</i>	47
Figura 23 – Diagrama de Exemplo.	50
Figura 24 – Diagrama de Exemplo após a Transformação das Classes.	51
Figura 25 – Diagrama de Exemplo Após o Mapeamento das Generalizações.	52
Figura 26 – Diagrama de Exemplo Após o Mapeamento dos Atributos Temporais.	53
Figura 27 – Diagrama de Exemplo Após o Mapeamento das Associações.	54
Figura 28 – Diagrama de Exemplo com Composição.	55
Figura 29 – Diagrama de Exemplo com Agregação.	56
Figura 30 – Código Gerado pelo Diagrama com Associação Simples.	57
Figura 31 – Código Gerado pelo Diagrama com Agregação.	57
Figura 32 – Código Gerado pelo Diagrama com Composição.	58
Figura 33 – <i>Feedback</i> Final.	58

LISTA DE TABELAS

Tabela 1 – Itens Básicos do Diagrama de Classes.	41
Tabela 2 – Propriedades Específicas de Cada Item.	41
Tabela 3 – Exceções da <i>GenJPA</i>	44
Tabela 4 – Classes de Controle.	49
Tabela 5 – Anotações Suportadas pela <i>GenJPA</i>	50
Tabela 6 – Comparativo entre Ferramentas	59

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programing Interface</i>
CASE	<i>Computer Assisted Software Engineering</i>
CIM	<i>Computation Independent Model</i>
EA	<i>Enterprise Architect</i>
GUI	<i>Graphical User Interface</i>
GWT	<i>Google Web Toolkit</i>
IDE	<i>Integrated Development Environment</i>
JPA	<i>Java Persistence API</i>
JVM	<i>Java Virtual Machine</i>
MTL	<i>Model To Text Transformation Language</i>
MDA	<i>Model Driven Architecture</i>
MD	<i>Model Driven</i>
MDD	<i>Model Driven Development</i>
MDE	<i>Model Driven Engineering</i>
MOF	<i>Meta Object Facility</i>
OMG	<i>Object Management Group</i>
OO	<i>Orientação a Objetos</i>
ORM	<i>Object/Relational Mapping</i>
PSM	<i>Platform Specific Model</i>
UML	<i>Unified Modeling Language</i>
XMI	<i>XML Metadata Interchange</i>
XML	<i>Extensible Markup Language</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	Contextualização	14
1.2	Justificativa	15
1.3	Discussão	17
1.4	Objetivos	17
1.4.1	Objetivo Geral	17
1.4.2	Objetivos Específicos	17
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	Orientação a Objetos	19
2.1.1	Classes e Objetos	19
2.1.2	Heranças e Polimorfismo	20
2.2	UML	20
2.2.1	Conceitos	21
2.2.2	Diagrama de Classes	22
2.2.3	Perspectiva de Dados no Diagrama de Classes	23
2.3	Engenharia à Frente	23
2.3.1	Conceitos	24
2.3.2	Metodologias Dirigidas por Modelos	25
2.3.3	<i>Model Driven Architecture</i>	26
2.4	Java	27
2.4.1	Conceitos	27
2.4.2	Tipos de Dados	29
2.5	Mapeamento Objeto-Relacional	29
2.6	Influência da JPA nos Itens de Interesse do Modelo	30
2.7	Classe	30
2.7.1	Atributo da Classe	31
2.8	Relacionamentos	32
2.8.1	Generalização	32
2.8.2	Associação	32
2.8.3	Agregação e Composição	34
3	TRABALHOS RELACIONADOS	35
3.1	<i>GenCode</i>	35
3.2	<i>HiberObjects</i>	36
3.3	<i>MD-JPA</i>	36

4	GENJPA	38
4.1	Módulo de Extração	40
4.1.1	Normalização de Nomes	42
4.1.2	Resultados da Etapa de Extração	42
4.2	Fase de Validação	42
4.2.1	Interpretação do Arquivo XML	43
4.2.2	Validação dos Objetos	44
4.2.2.1	Validação dos Nomes	45
4.2.2.2	Validação dos Tipos	46
4.2.2.3	Validação das Generalizações	46
4.2.2.4	Validação das Associações	46
4.2.3	Finalização das Validações	47
4.3	Fase de Construção	48
4.3.1	Módulo de Controle	48
4.4	Mapeamento objeto-relacional	49
4.4.1	Transformação de Classes em Entidades	50
4.4.2	Mapeamento de Heranças	51
4.4.3	Mapeamento de Atributos Temporais	52
4.4.4	Mapeamento de Associações	53
4.4.4.1	Mapeamento de Composição	54
4.4.4.2	Mapeamento de Agregação	55
4.5	Geração do Código Executável	56
5	CONCLUSÃO	59
5.1	Trabalhos Futuros	60
	REFERÊNCIAS	61
	ANEXO A – REGRAS DE EXTRAÇÃO EM ACCELEO	63
	ANEXO B – SISTEMA DE LEILÃO ONLINE	65
B.1	Diagrama de Classes	65
B.2	Entidades	65
B.2.1	Participante.java	65
B.2.2	Lance.java	67
B.2.3	Leilao.java	69
B.2.4	Item_Leilao.java	71
B.3	Diagrama Entidade-Relacionamento	73
	ANEXO C – SISTEMA BANCÁRIO	74

C.1	Diagrama de Classes	74
C.2	Entidades	74
C.2.1	Conta_Comum.java	74
C.2.2	Conta_Poupanca.java	77
C.2.3	Conta_Especial.java	78
C.2.4	Pessoa.java	79
C.2.5	Pessoa_Juridica.java	81
C.2.6	Pessoa_Fisica.java	81
C.2.7	Movimento.java	82
C.3	Diagrama Entidade-Relacionamento	84

1 INTRODUÇÃO

Nesse capítulo são apresentadas as razões que levaram a criação da ferramenta. Apresenta-se também alguns dos conceitos introdutórios que fazem parte do domínio da aplicação, além de linguagens e ferramentas.

1.1 CONTEXTUALIZAÇÃO

A modelagem é uma técnica de engenharia aprovada, bem aceita e utilizada por engenheiros para descrever e planejar seus projetos. Na indústria de *software*, modelos são utilizados na especificação e visualização dos diversos itens que compõe a estrutura e comportamento de um sistema de informação, bem como na documentação das decisões tomadas durante seu desenvolvimento [3]. Um modelo é uma simplificação da realidade, ou seja, uma abstração do domínio do problema que descreve o sistema sob determinada perspectiva. Eles podem abranger planos detalhados ou simplificados do projeto, possuindo sintaxe e semântica distinta que os diferencia [8].

Assim como na arquitetura e na engenharia, onde existem métodos padrões para realização da modelagem de projetos, também existe uma linguagem padrão para a modelagem de sistemas de informação, mais especificamente daqueles que seguem o paradigma orientado a objetos: a Linguagem Unificada de Modelagem (UML) (do inglês *Unified Modeling Language*). A UML é uma linguagem de modelagem visual que estabelece vocabulário e regras específicos para construção de modelos que apresentem a estrutura do projeto e forneçam uma visão conceitual e física do sistema [3]. Ela define um conjunto de blocos de construção e itens relacionais que permitem ao projetista criar modelos, em formato de diagrama, que apresentem uma ou mais visões do sistema projetado.

Na UML, os diagramas são classificados em dois grupos de acordo com o tipo de visão fornecida por seus componentes [1]: estruturais e comportamentais. Diagramas estruturais têm seu foco voltado para representação e visualização dos componentes estáticos do sistema, ou seja, aqueles que estão menos propensos a mudanças, como classes, interfaces e pacotes. Já diagramas comportamentais dão ênfase à dinâmica do *software*, descrevendo suas rotinas e funcionamento interno. Quando agrupados, modelos estruturais e comportamentais fornecem uma ampla visão das características do sistema.

O diagrama de classes é o mais utilizado e um dos mais importantes da UML [8]. Nele, está definido o principal bloco usado na implementação de sistemas orientados a objetos: a classe [3]. Uma classe identifica e caracteriza determinado elemento de interesse da aplicação, apresentando suas propriedades e comportamento. Por meio de classes, itens conceituais e concretos, que formam o domínio do problema da aplicação, são identificados, fornecendo assim

uma visão estrutural da arquitetura do sistema.

Todas as linguagens de programação orientada a objetos fornecem notações para o conceito de classe. Por isso, é comum que o diagrama citado seja utilizado como artefato de entrada para o processo de engenharia à frente [3]. Essa técnica, implementada por ferramentas específicas, consiste na geração de código a partir de modelos detalhados.

Utilizando regras de transformação que relacionam abstrações do modelo à elementos de uma determinada linguagem de programação, tais ferramentas de transformação conseguem automatizar a implementação do sistema. Nesse processo, os conceitos de classe, atributo, operação, relacionamento, dentre outros do modelo de classes são usados para gerar a definição básica da aplicação, isto é, seu esqueleto.

A maioria das ferramentas *CASE* do mercado são capazes de executar a engenharia à frente a partir de diagramas de classes, porém muitas apresentam falhas ao fazer o mapeamento de alguns elementos mais complexos da UML, seja devido a limitações da linguagem de destino ou por desconsiderarem a existência deles no diagrama [1]. Conceitos importantes do diagrama de classes, como o de agregação e composição das associações, não são mapeados para o código por nenhuma das principais ferramentas do mercado [1] [9].

A constatação de tal problema serviu de motivação para elaboração da ferramenta apresentada nesta monografia, a *GenJPA*. Nela foram implementadas regras de mapeamento mais específicas capazes de transcrever ao código os conceitos que outras ferramentas deixam de mapear.

1.2 JUSTIFICATIVA

A geração de código Java a partir de diagramas da UML, além de ser tema de estudo no meio acadêmico é prática recorrente no mercado de *software*. Hoje, muitas ferramentas, dentre elas as três maiores do mercado de modelagem: *Astah*, *Enterprise Architect* (EA) e *Rational Software Architect* (RSA), fornecem suporte à geração de código Java [1]. Todavia, como descreve Magalhães [1], observa-se inconsistências no código gerado por estas aplicações.

Causada, principalmente, pela falta de regras específicas de mapeamento para itens mais complexos da UML, a inconsistência ocorre quando determinado conceito do modelo deixa de ser transcrito para o código durante a engenharia à frente [1]. Nesse caso, por conta da perda de informações, o código gerado apresenta divergência com o projeto da aplicação e com isso algumas restrições do sistema deixam de existir.

Segundo Magalhães [1] e Gessenharter [9] as inconsistências mais comuns ocorrem durante a transformação de diagramas de classes, estando relacionadas aos conceitos de agregação, composição e navegabilidade das associações. Ambos os autores, após analisarem o processo de engenharia à frente UML-Java executado por ferramentas de transformação, chegaram as mesmas conclusões:

- Agregações e composições sempre são mapeadas para o código como associações simples, havendo perda de informações (Figura 1).
- A navegabilidade algumas vezes não é levada em consideração durante a transformação (Figura 2).

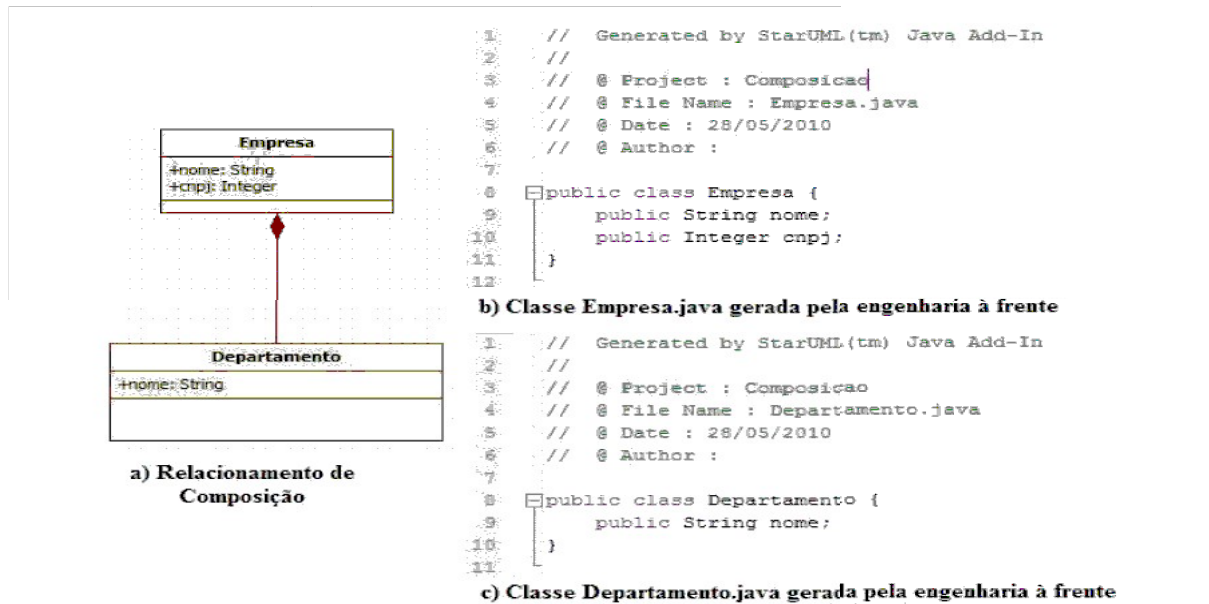


Figura 1 – Transformação Inconsistente de uma Composição [1].

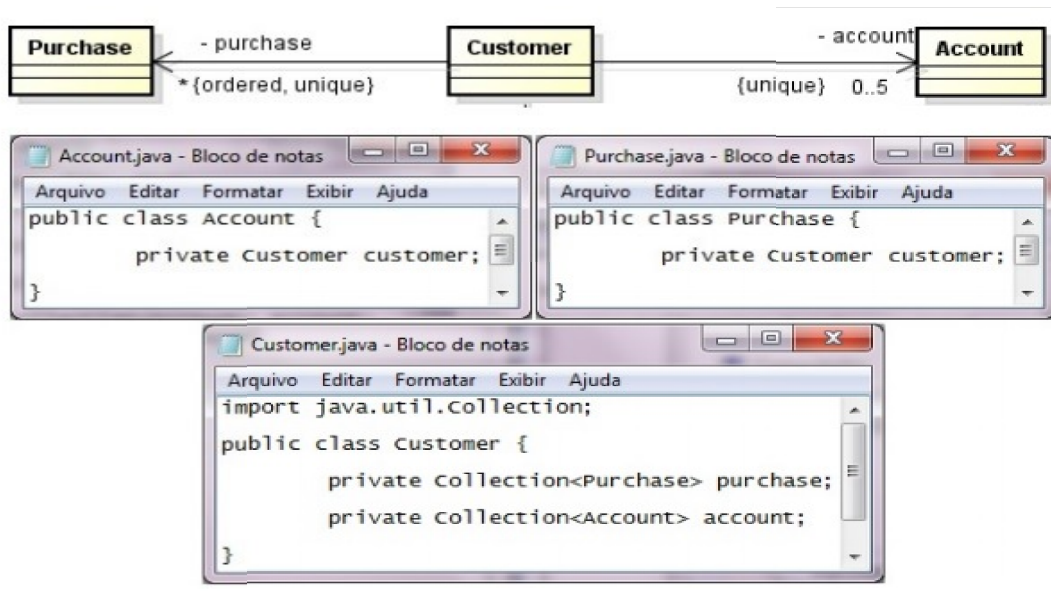


Figura 2 – Transformação Inconsistente da Navegabilidade [1].

Além da inconsistência, Magalhães [1] apresenta também a deficiência das ferramentas *Astah*, *EA* e *RSA* no que tange a Interação Humano Computador (IHC). Segundo o autor, os

mecanismos de interação fornecidos por todas elas são mínimos, limitando-se à exibição de mensagens de erro. Além disso, estas ferramentas obrigam o usuário a trabalhar conforme suas convenções de mapeamento, não oferecendo qualquer opção para configuração da transformação.

1.3 DISCUSSÃO

Alguns dos problemas apresentados podem ser solucionados dando mais atenção à etapa de transformação, casos da navegabilidade e interação com o usuário. Por outro lado, levando em consideração que os conceitos de agregação e composição não possuem uma representação correspondente na linguagem Java, torna-se necessário o uso de tecnologias auxiliares, cuja notação forneça abstrações capazes de mapear a semântica desses itens para o código.

O mapeamento objeto-relacional consegue transportar para o código os conceitos mais complexos da associação do diagrama de classes da UML. Por isso, na *GenJPA*, foi usada uma abordagem mista baseada nesse paradigma. Nela, código Java em conjunto com anotações e propriedades da *Java Persistence API* (JPA) são usadas no mapeamento de agregações e composições da UML. Com isso conseguiu-se diminuir a inconsistência e manter as restrições do modelo.

O processo de engenharia à frente executado pela aplicação gera, de forma automática, as entidades de um sistema Java. Essas classes trabalham a nível objeto-relacional, intermediando a troca de informações entre o banco de dados e a aplicação.

1.4 OBJETIVOS

1.4.1 Objetivo Geral

Implementar uma ferramenta de engenharia à frente que seja capaz de mapear ao código conceitos mais complexos do diagrama de de classes (agregações, composições e navegabilidade) para linguagem Java/JPA. Fornecendo ao usuário mecanismos de interação (GUI) capazes de alterar o comportamento padrão da aplicação.

1.4.2 Objetivos Específicos

- Identificar os componentes de interesse do diagrama de classes e suas variações sintáticas;
- Implementar uma interface gráfica, onde o projetista possa repassar à aplicação informações que influenciarão o código gerado;
- Desenvolver o módulo capaz de converter blocos da UML para suas respectivas representações em Java / JPA;

CONSIDERAÇÕES FINAIS

Os próximos capítulos estão organizados da seguinte forma:

- **Fundamentação Teórica:** Nesse capítulo são apresentados todos os conceitos teóricos utilizados no desenvolvimento da solução proposta;
- **Trabalhos Relacionados:** Nesse capítulo são apresentados trabalhos cujo conteúdo descreve ferramentas de transformação que implementam a engenharia à frente a partir de diagramas de classes tendo a linguagem Java como destino;
- **GenJPA:** Capítulo dedicado a apresentação da solução implementada, detalhando funcionalidades, módulos e implementação;
- **Conclusão:** Nesse capítulo é feita a conclusão do trabalho dado seus objetivos propostos e são listados os trabalhos futuros para melhorar e/ou expandir a utilização da solução.

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo são apresentados todos os conceitos relacionados à implementação da ferramenta proposta. Aqui são descritos os paradigmas, conceitos e tecnologias que serviram de base para elaboração da *GenJPA*.

2.1 ORIENTAÇÃO A OBJETOS

Hoje, grande parte dos sistemas de informação são desenvolvidos seguindo um mesmo paradigma de programação, que se mostra superior à antiga visão baseada em algoritmos [3]: o orientado a objetos (OO). Esse padrão utiliza os conceitos de classificação e abstração na representação do conhecimento, permitindo que as informações sejam estruturadas de uma maneira mais próxima ao pensamento humano [8].

Tanto o Java quanto a UML têm suas sintaxes sedimentadas nos itens da orientação a objetos. Por esta razão, se faz necessário compreender os principais conceitos desse paradigma antes de apresentar o funcionamento das duas linguagens.

2.1.1 Classes e Objetos

A visão orientada a objetos, diferentemente do método procedural, não se limita apenas à conceitos do algoritmo (funções e rotinas), ela define maneiras mais sofisticadas para estruturar as informações, através das classes e dos objetos.

Na orientação a objetos, uma classe abstrai a notação de determinado conceito geral [10]. Ela pode ser vista como um molde ou uma planta de projeto, usada para dar origem a instâncias reais do conceito definido [3]. Utilizando classes, qualquer elemento do mundo real pode ser abstraído para uma representação processável por máquina.

A classe define uma série de propriedades, chamadas de atributos, cujos valores variam de uma instância a outra. Além de atributos, a classe pode definir métodos, conhecidos também como operações capazes de executar determinada ação ou mudar o estado dos objetos [10]. Normalmente, essas operações contêm parâmetros usados como entrada para o seu processamento.

A Figura 3 traz a representação dos conceitos de classes e objetos. Nela, a classe Pessoa, é usada na instanciação de três objetos: João, Maria e Carlos.

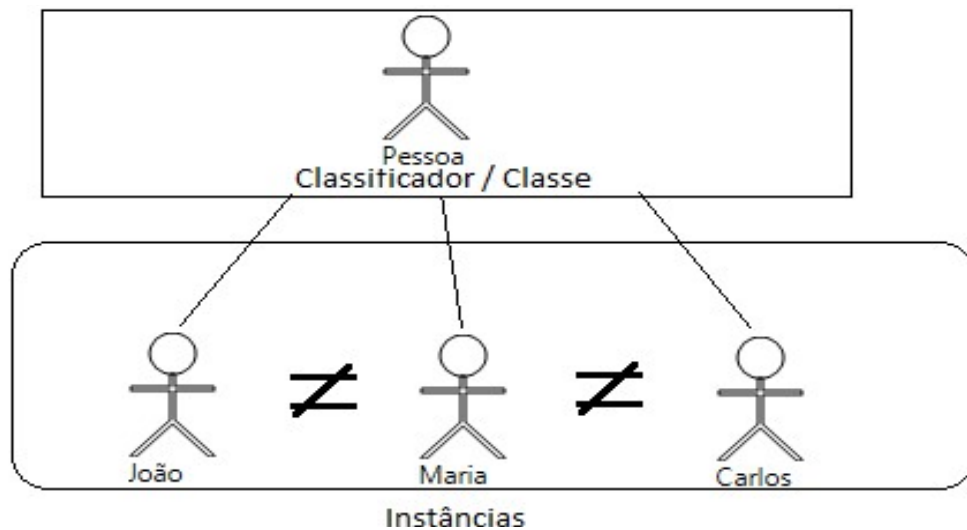


Figura 3 – Representação de Classes e Objetos.

2.1.2 Heranças e Polimorfismo

No mundo real, algumas abstrações são derivadas de outras, levando isso em consideração o paradigma OO define o relacionamento de polimorfismo e herança, tido como uma de suas características mais poderosas [8].

Heranças trabalham com os conceitos de superclasse e subclasse. Nesse tipo de relacionamento, uma classe geral é utilizada para representar todas as propriedades e ações compartilhadas por seus diversos filhos. Na herança, classes filhas herdam todas as características da sua classe mãe, podendo ainda definir atributos e operações adicionais [3].

Um caso especial de herança, baseada nos conceitos de tipo e subtipo origina outro conceito da orientação a objetos: polimorfismo. Segundo Junior [10], o polimorfismo ocorre quando uma classe filha reescreve algum método de sua classe mãe, modificando o comportamento padrão dessa operação. Assim, no momento da invocação do método, diferentes implementações podem ser executadas.

2.2 UML

Sistemas de informação caracterizam-se por sua dinamicidade, isto é, a propriedade que estes possuem de crescer com o decorrer do tempo, seja devido a mudanças no negócio ou por erros na análise inicial do projeto. Geralmente, são elementos de grande complexidade, compostos por diversos itens, cuja arquitetura depende de inúmeros fatores [8]. Por mais simples que um *software* possa parecer inicialmente, uma série de pontos como requisitos, custo de produção, manutenções e reusabilidade devem ser definidos antes de seu desenvolvimento.

A UML fornece os mecanismos necessários para especificação, visualização, construção e documentação da arquitetura desses sistemas complexos [3].

2.2.1 Conceitos

A UML (*Unified Modeling Language*) ou Linguagem Unificada de Modelagem é a notação padrão da indústria de *software* para modelagem de sistemas de informação orientados a objetos [8]. Booch e outros [3] a definem como uma linguagem de modelagem visual de grande abrangência, cuja sintaxe fornece todos os meios necessários para modelagem e planejamento dos mais diversos tipos de sistemas de informação. Em sua notação, a linguagem define uma série de regras e componentes, agrupados em diagramas, que em conjunto fornecem uma projeção do sistema que se está desenvolvendo ou que se pretende desenvolver.

Três tipos de blocos formam o vocabulário da UML: itens, relacionamentos e diagramas [3]. Itens são cidadãos de primeira classes usados para representar as características estáticas e dinâmicas do sistema. Relacionamentos são os elementos que conectam estes itens, modelando as colaborações entre partes. Por fim, os diagramas são junções desses dois conceitos, cuja função é fornecer uma visão de determinada característica do sistema.

Diagramas da UML proveem abstrações que facilitam a especificação e visualização dos muitos artefatos que fazem parte da arquitetura de um sistema de informação [8]. Eles representam simplificações da realidade, que auxiliam a equipe de desenvolvimento no decorrer do projeto [3]. Dentro do ciclo de desenvolvimento, diagramas são usados para guiar a implementação do sistema, bem como para documentar as decisões tomadas.

Ao todo, a UML define 14 tipos de diagramas divididos em dois grupos (Figura 4). Essa classificação se dá de acordo com a visão fornecida pelo modelo e dos itens nele contidos.

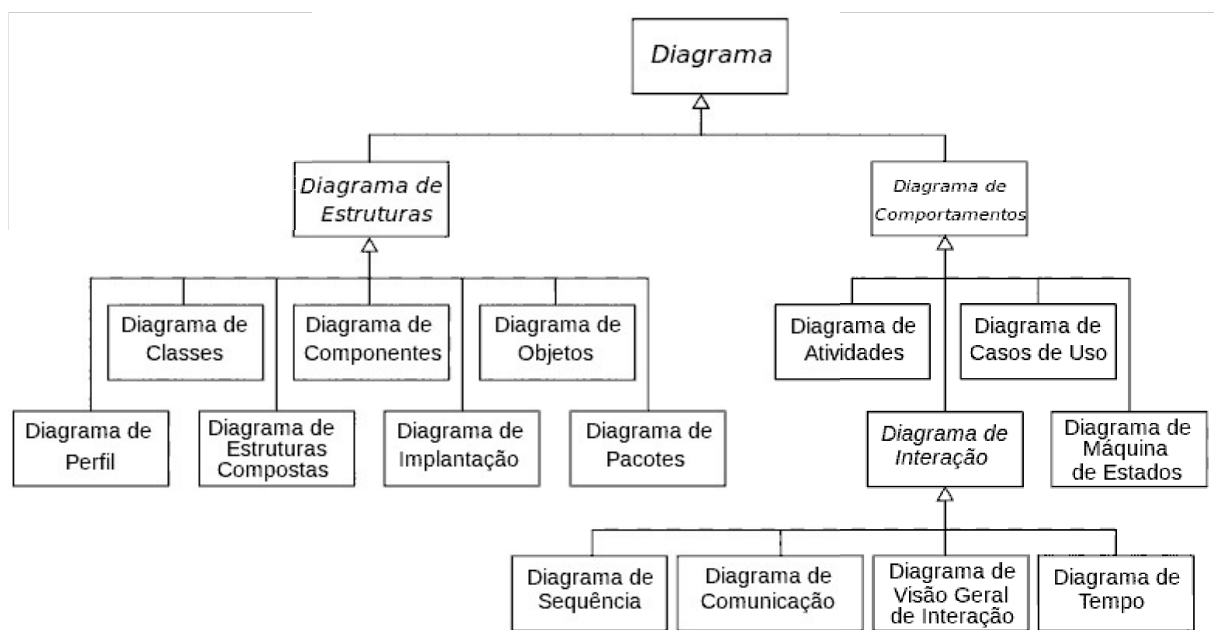


Figura 4 – Organização dos Diagramas na UML. Adaptado da *UML Superstructure* [2].

Diagramas estruturais são usados para visualização, especificação, construção e documentação de aspectos estáticos do sistema, isto é, sua estrutura relativamente estável, cujos itens estão menos propensos a mudanças. Já os diagramas comportamentais modelam os aspectos dinâmicos do *software*, suas partes que sofrem influência do fator tempo, como as rotinas e interações [1].

Segundo estudo realizado por Dobing e Parson [11], a frequência de utilização de alguns diagramas da UML é superior a outros. Modelos de casos de uso, sequência e classes se fazem mais presentes nos projetos. Dos três, o diagrama de classes é o mais utilizado, apresentando uma taxa de uso superior a 70%. Diversos autores afirmam que esse diagrama é o mais importante da linguagem [3] [8] [1].

2.2.2 Diagrama de Classes

O diagrama de classes é o mais utilizado e também o que se encontra com maior frequência em projetos de *software* orientados a objetos [8]. Esse diagrama exhibe um conjunto de classes, interfaces, colaborações e relacionamentos, sendo capaz de abstrair o projeto arquitetural do *software* [3]. Linguagens de programação orientadas a objetos, como Java, C++ e Visual Basic apresentam em suas sintaxes abstrações relativas a muitos dos conceitos deste modelo.

O bloco de construção central do diagrama de classes é a classe. Usada para representar as diversas abstrações que fazem parte do domínio do problema, uma classe descreve um conjunto de elementos que compartilham as mesmas características, comportamentos, relacionamentos e significado [3]. Por meio de colaborações, classes se conectam e executam as rotinas da aplicação [8].

Conforme o nível de detalhamento apresentado, os diagramas de classes são classificados em três tipos [12]:

- **Conceituais:** são diagramas de classes cuja função é representar os relacionamentos e itens básicos do sistema. Na perspectiva conceitual, preocupa-se em descrever, na forma de classes, as abstrações mais importantes do *software* e as associações essenciais estabelecidos entre elas. A perspectiva conceitual fornece uma visão geral dos conceitos que fazem parte do domínio do problema.
- **Projeto:** são diagramas que, além de apresentar os principais conceitos e relacionamentos do sistema, definem o comportamento das abstrações. Na perspectiva de projeto, trabalha-se com o conceito de responsabilidade da classe, o qual define sua função dentro do sistema. Diagramas desse tipo fornecem uma visão ampla da estrutura das classes e de suas operações.
- **Implementação:** nessa perspectiva o diagrama de classes apresenta completude, isto é, contém todas as classes que compõem o *software*, bem como o detalhamento dos relacionamentos. Diagramas de classes modelados conforme a perspectiva de

implementação estão conectados às tecnologias específicas (linguagem de programação).

2.2.3 Perspectiva de Dados no Diagrama de Classes

No contexto desse trabalho, cabe citar um caso especial do uso de diagramas de classes, que ocorre quando os mesmos são utilizados na modelagem de classes persistentes do sistema. Estas classes, também chamadas de entidades, possuem a característica de gerar objetos cujos dados transcendem o tempo de execução da aplicação [8].

Classes persistentes estão relacionadas às tabelas de banco de dados relacionais e seus objetos aos registros destas tabelas. O diagrama de classes, quando utilizado na modelagem desse tipo de item, fornece uma visão estrutural do esquema lógico do banco de dados da aplicação [8]. Diagramas projetados conforme essa visão são usados na criação das tabelas de banco de dados relacionais, bem como na geração de entidades em sistemas orientado a objetos.

A Figura 5 mostra um diagrama de classes modelado conforme a perspectiva de dados. Todas as classes do modelo geram instâncias cujos dados precisam ser armazenados para posterior consulta.

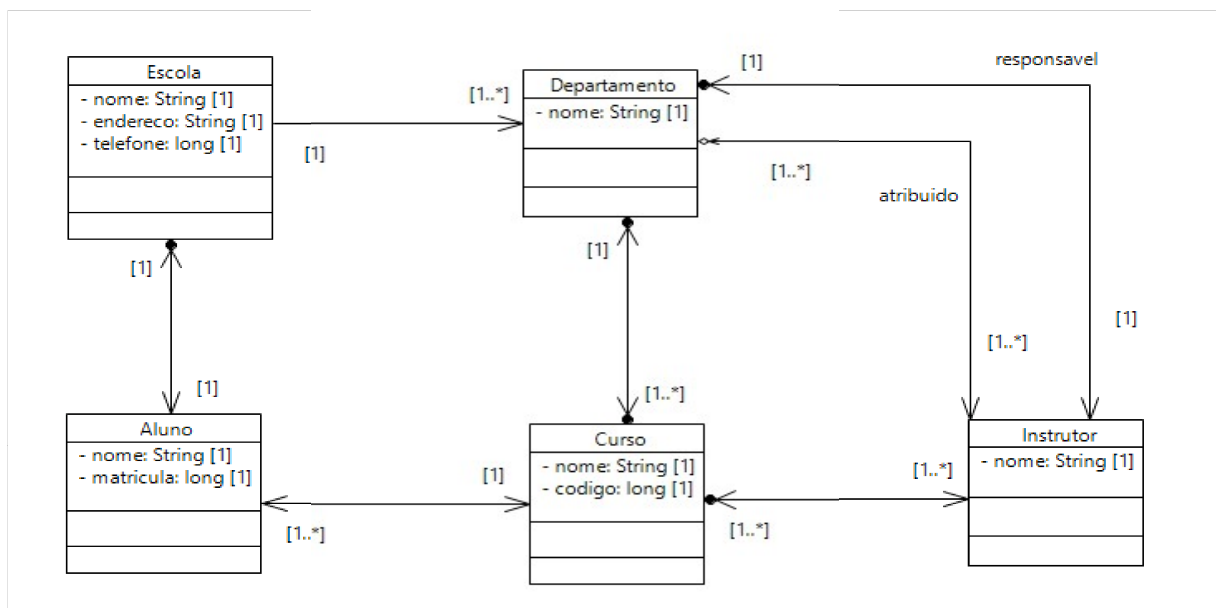


Figura 5 – Diagrama de Classes que Apresenta Entidades. Adaptado de Booch e outros [3].

2.3 ENGENHARIA À FRENTE

Os modelos do *software* são componentes de grande valor dentro do projeto, no entanto, eles não são os artefatos de maior importância produzidos durante o desenvolvimento de um sistema [3]. Todo o esforço empenhado pelos diversos membros da equipe, nas várias etapas que compõem o ciclo de desenvolvimento de um sistema de informação, têm um único objetivo:

entregar ao cliente uma aplicação funcional capaz de atender às necessidades de seu negócio a partir da informatização de rotinas antes não assistidas por computador.

Linguagens de modelagem como a UML auxiliam os desenvolvedores a produzir uma implementação correta do *software* [8]. Diagramas como o de classes e o de sequência, por exemplo, possuem clara conexão com o código, podendo ser usados na etapa de implementação. Quando uma ferramenta é utilizado para interpretar esses modelos e gerar o código executável caracteriza-se a engenharia à frente.

Esta seção apresenta os principais conceitos por trás desse processo, seus benefícios e as metodologias de desenvolvimento de software que o utilizam.

2.3.1 Conceitos

Além de ser uma linguagem para documentação, especificação e visualização de projetos de sistemas OO, a UML é também uma linguagem para construção, isto é, a partir de seus diagramas é possível obter uma implementação funcional do sistema ou de parte dele. Segundo Booch e outros [3], a UML foi projetada com a finalidade de ser mapeada para várias linguagens de programação, ou seja, muitos dos conceitos contidos em seus diagramas podem ser diretamente conectados a estas linguagens.

A interpretação de diagramas da UML pode ser realizada da maneira tradicional, pelo programador, ou de forma automática, utilizando regras de transformação e ferramentas específicas [13]. No segundo caso, quando sistemas são usados para gerar sistemas, dá-se o nome de engenharia à frente.

A engenharia à frente é o processo de transformação de um modelo em código, implementado por ferramentas específicas de mapeamento [14]. Essas aplicações definem regras capazes de transportar componentes da UML para alguma linguagem de programação orientada a objetos, transcrevendo os conceitos do diagrama para construções no código de mesmo valor. As ferramenta que implementam a engenharia à frente são classificadas de acordo com a técnica de mapeamento empregada, podendo ser do tipo *Template* ou Programas Geradores de Código [14].

Ferramentas *Template* fazem um mapeamento direto entre modelo e texto por meio *engines* de transformação. Esse mapeamento é do tipo um para um, onde cada conceito do modelo é conectado a um elemento do código. Por outro lado, programas geradores de código implementam regras próprias de validação, mapeamento e transformação, estando voltadas para domínios específicos de aplicação. Programas geradores de código costumam apresentar uma engenharia à frente mais completa, cujo código gerado tende a ser mais consistente. [13].

No campo da engenharia de *software*, ambos os tipos de ferramentas de transformação servem de apoio para as metodologias dirigidas por modelos.

2.3.2 Metodologias Dirigidas por Modelos

Muitas vezes os prazos que se dispõe para a entrega do sistema é curto, mas, por outro lado, sua complexidade é alta. Como consequência ocorre um aumento no custo de produção, já que mais pessoas precisarão estar envolvidas no projeto para que se cumpra o prazo estabelecido. Levando em consideração esse tipo de cenário surgiram as metodologias dirigidas por modelos (MDs) (do inglês *Model Driven*): abordagens que aplicam a técnica de engenharia à frente com o objetivo de automatizar a construção do *software* e assim diminuir os custos de produção e tempo de entrega [14].

Segundo Magalhães [1], nas MDs os diagramas são cidadãos de primeira classe, usados não apenas na documentação do *software*, mas também para obtenção de uma versão executável do mesmo. Modelos abstratos, criados durante o projeto, são enriquecidos até um ponto que se possa obter uma implementação funcional do sistema.

Nessas metodologias, há uma preocupação maior com relação à modelagem, gasta-se mais tempo durante essa atividade para que os diagramas produzidos sejam os mais completos. Todavia, o tempo gasto construindo tais modelos é compensado com o tempo que seria gasto na implementação de várias linhas de código [1].

Hoje, três abordagens se destacam na indústria, sendo uma delas especificada pelo órgão que rege a UML, o Grupo de Gerenciamento de Objetos (OMG) (do inglês *Object Management Group*) [1]. São elas:

- ***Model Driven Development (MDD)***: abordagem que defende o uso de modelos em todas as etapas do ciclo de desenvolvimento. Nela, os diagramas que abstraem as interfaces, interações e propriedades da aplicação, além de usados na análise, servem para automatizar a implementação.
- ***Model Driven Architecture (MDA)***: visão particular do *Object Management Group* sobre a MDD. Na MDA, a UML é tida como uma linguagem de programação. Nessa metodologia, são estabelecidas uma série de tecnologias que auxiliam a construção de novas ferramentas de transformação. A abordagem defende, além das transformações do tipo modelo-código aquelas do tipo modelo-modelo.
- ***Model Driven Engineering (MDE)***: metodologia que engloba as outras duas apresentadas. Nessa abordagem, a construção do sistema pode ser totalmente automatizada. Para isso, utilizam-se ferramentas de transformação capazes de gerar código estrutural e comportamental a partir de diagramas de classes e sequência.

Por utilizar padrões estabelecidos pelo OMG e conectar-se com a UML, a MDA foi tomada como referência na implementação da ferramenta apresentada nesta monografia. A próxima seção descreve de forma mais detalhada os conceitos da metodologia.

2.3.3 Model Driven Architecture

A *Model Driven Architecture* é a abordagem padrão da *Object Management Group*, órgão que normatiza a UML, para se trabalhar com o desenvolvimento dirigido por modelos. Tal abordagem defende o uso de diagramas desde as primeiras etapas do projeto, bem como a utilização destes na automatização das etapas seguintes do desenvolvimento.

Na MDA, os modelos de *software* são agrupados em três tipos de acordo com o nível de detalhamento apresentado: CIM, PIM e PSM [15]. Para compreender essa classificação, pode-se levar em consideração as perspectivas de modelagem dos diagramas de classes.

Traçando um paralelo, o modelo computacional independente (CIM) (do inglês *Computation Independent Model*) está relacionado à diagramas de classe modelados a partir da perspectiva conceitual, que apresentam os principais conceitos do sistema e possuem alto nível de abstração. Já o modelo independente de plataforma (PIM) (do inglês *Platform Independent Model*) se conecta aos diagramas modelados conforme a perspectiva de projeto, que apresentam as propriedades das classes sem focar na implementação das operações. Por fim, o modelo específico de plataforma (PSM) (do inglês *Platform Specific Model*) se conecta com diagramas de classes modelados sob a perspectiva de implementação, cujo conteúdo apresenta alto nível de detalhamento, sendo descritos para uma plataforma específica.

Segundo Maia [15], a MDA consiste na definição inicial dos requisitos através do CIM, a transformação deste CIM para um modelo independente de plataforma (PIM), utilizando uma linguagem de modelagem, e a preparação deste modelo independente para construção de um PSM, através do processo de engenharia à frente. A Figura 6 demonstra esse procedimento.

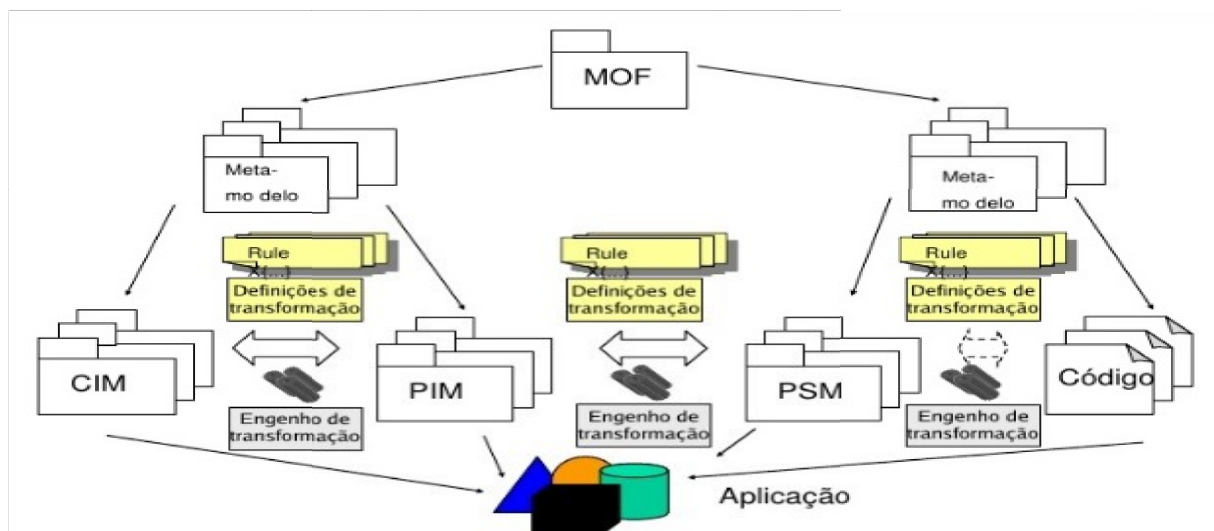


Figura 6 – Model Driven Architecture [4].

Modelos específicos de plataforma (PSM) são usados na realização da engenharia à frente na MDA [1]. Por ser uma especificação da OMG, os conceitos que envolvem essa transformação

estão voltados quase que completamente para diagramas da UML.

A MDA estabelece uma série de padrões que ajudam na construção de ferramentas que apoiem o processo de engenharia à frente. Três deles foram utilizados na implementação da *GenJPA*.

- **Meta Object Facility (MOF):** linguagem usada para especificar formalmente linguagens de modelagem. Com o MOF é possível definir os blocos que fazem parte destas linguagens através de elementos estruturais como classes, atributos e operações. O metamodelo da UML é especificado usando essa linguagem. Nele são definidos quais elementos fazem parte de cada um dos diagramas e quais as propriedades e relacionamentos que podem ser estabelecidos entre estes itens. Segundo Magalhães [1], linguagens definidas através da MOF podem ser mapeadas umas para as outras, facilitando o processo de engenharia à frente e as transformações modelo-modelo.
- **XML Metadata Interchange (XMI):** define um conjunto de regras que mapeiam os diagramas baseados em MOF para documentos no formato XML. Através dessa especificação, diagramas da UML podem ser representados em formato estruturado permitindo assim interoperabilidade entre ferramentas de transformação e CASE [16]. O formato XMI provê uma facilitação para troca de modelos e uma padronização para sua representação.
- **Model to Text Transformation Language (MTL):** linguagem utilizada para extração de dados de diagramas da UML em formato XMI. Baseia-se no metamodelo da UML e tem como função a transformação de diagramas em artefatos no formato texto, como código, especificações de implantação, relatórios e documentos.

2.4 JAVA

Dentre as ferramentas de modelagem (CASE) e transformação há uma linguagem de programação que recebe maior suporte, resultado, principalmente, da sua ampla difusão no mercado de desenvolvimento. Dados retirados do site da linguagem, que na realidade trata-se de um plataforma, comprovam seu sucesso comercial, especialmente, em ambiente web e mobile. Com uma base operacional de 3 bilhões de dispositivos, dentre computadores, celulares e microcontroladores, o Java é hoje a principal linguagem de desenvolvimento do mercado [17].

2.4.1 Conceitos

A linguagem Java engloba os diversos conceitos do paradigma orientado sendo baseada no C++ [18]. Assim como o C++, a sintaxe do Java trabalha com os conceitos de classes, interfaces, operações, atributos, herança e polimorfismo, todavia, não incorporando a herança múltipla, o gerenciamento manual de memória e tratamento obrigatório de exceções [19]. O Java tem forte apelo para o ambiente web, pois permite que os componentes heterogêneos de uma

rede se comuniquem de forma transparente independentemente das suas arquiteturas, sistemas operacionais e tipos de dispositivo [20].

Buscando segurança e principalmente a portabilidade do *software*, a plataforma Java utiliza o conceito de máquina virtual na execução de seus programas [20]. A Máquina Virtual Java (JVM) (do inglês *Java Virtual Machine*) trabalha como intermediador entre aplicação e o hardware do equipamento, interpretando o código e fornecendo as instruções para as camadas específicas do sistema operacional que devem executá-las [21]. Para cada sistema operacional, o Java fornece uma implementação da JVM, sendo esta responsável por interpretar as instruções do programa e repassá-las ao equipamento [20].

Segundo Braz [20], essa característica possibilita que sistemas escritos com a linguagem possam ser executados em qualquer arquitetura de hardware e sistema operacional sem a necessidade de recompilação. Programas Java não são diretamente compilados para linguagem de máquina, mas sim para um código intermediário (*Bytecode*) interpretável por qualquer implementação da JVM. Desse modo, uma mesma aplicação pode rodar em diferentes sistemas operacionais, oferecendo as mesmas funcionalidades. A Figura 7 mostra o processo de compilação de um programa Java.

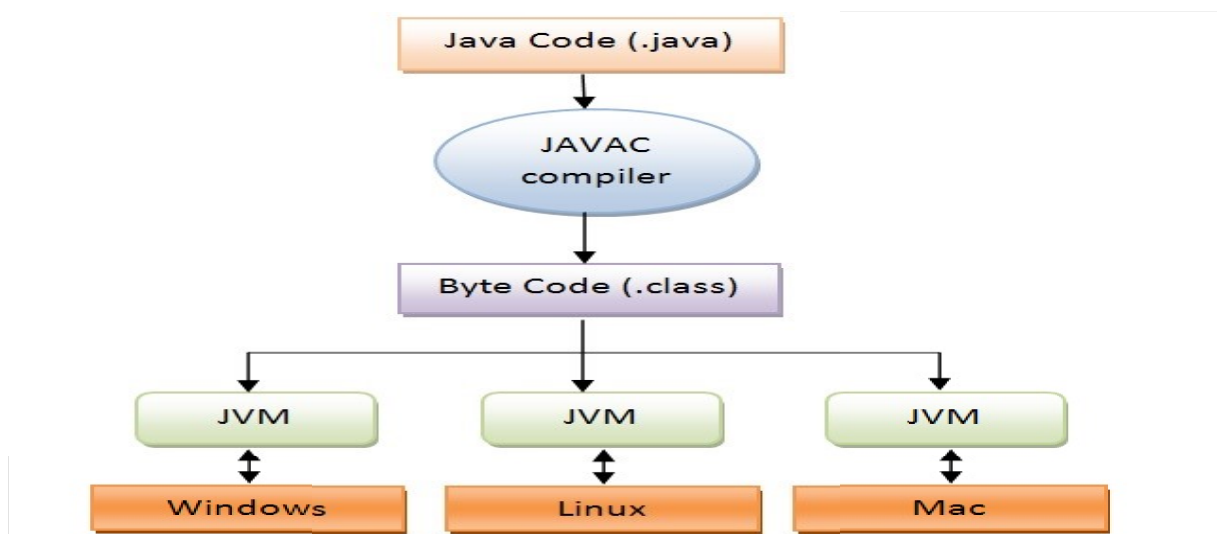


Figura 7 – Execução de um Programa Java. Adaptado de *Devmedia* [5].

Ao todo, a plataforma fornece três especificações (*Java Standard Edition*, *Java Enterprise Edition* e *Java Micro Edition*) as quais contém uma série de ferramentas e APIs (*Application Platform Interface*) usadas na construção de sistemas para ambiente *Web*, *Desktop* e *Mobile* [20]. Cada especificação engloba um conjunto específico de padrões, tecnologias e bibliotecas que têm o objetivo de agilizar a construção do *software*.

2.4.2 Tipos de Dados

O Java fornece uma infinidade de tipos de dados, isso porque trata-se de uma linguagem estaticamente tipada, onde as propriedades da classe têm seu tipo determinado no momento da criação [21]. Em seu pacote `java.lang` (raiz), a plataforma define seus tipos elementares, os quais podem ser utilizados na definição de qualquer atributo.

Tipos definidos no pacote `java.lang` são importadas automaticamente no momento da criação da classe. Já os externos a esse pacote precisam ser importados ou referenciados explicitamente nas classes que os utilizam.

2.5 MAPEAMENTO OBJETO-RELACIONAL

Apesar dos conceitos dos paradigmas orientado a objetos e relacional apresentarem equivalência a nível de modelo (perspectiva de dados), a nível de implementação o mapeamento de classes persistentes para banco de dados tende a ser mais complexo, isso porque a maneira que cada tecnologia estrutura seus dados é divergente, impossibilitando a troca direta de informações [7].

Enquanto banco de dados relacionais trabalham a nível de tabelas e registros, aplicações orientadas a objetos trabalham com classes e objetos. Essa divergência de conceitos torna necessária a ocorrência de transformações toda vez que algum dado trafegue da aplicação para o banco de dados ou do banco para a aplicação [7]. Para que uma classe Java consiga intermediar a comunicação entre banco e aplicação, se faz necessário um mecanismo que mapeie os conceitos dos dois modelos. A esse mecanismo dá-se o nome de mapeamento objeto-relacional (ORM) (do inglês *Object Relational Mapping*) [7].

Ferramentas de mapeamento objeto-relacional possuem recursos capazes de mapear classes persistentes para tabelas de base de dados relacionais, estabelecendo um caminho bidirecional para a troca de informações. Para isso, fazem uso de mecanismos de extensão e arquivos de configuração que estabelecem as regras a serem seguidas no processo de transformação de classes para tabela e de tabelas para classe.

Segundo Torres [7], ferramentas desse tipo abstraem totalmente a estrutura física do banco, permitindo que o desenvolvedor realize todas as operações (criação, consultas, alteração, inclusão e exclusão de dados) a nível de aplicação.

A plataforma Java, em sua especificação, define a *Java Persistence API* como o padrão para implementação de ferramentas de mapeamento-objeto relacional (ORM) [22]. A JPA estabelece o conjunto de regras, serviços e anotações básicas que devem ser fornecidos por uma ferramenta de mapeamento objeto-relacional.

Com JPA é possível mapear as classes da aplicação para o banco de dados, bem como seus atributos, heranças e associações. Para isso, são utilizadas anotações (mecanismos de

extensão) acoplados a esses elementos no código. A JPA consegue transportar todos os conceitos da orientação a objetos para representações equivalentes em banco de dados relacionais [22].

2.6 INFLUÊNCIA DA JPA NOS ITENS DE INTERESSE DO MODELO

Classes Java que representem entidades de banco de dados relacionais devem ser implementadas conforme a sintaxe *Bean* estabelecida na JPA [23]. Essa especificação define as anotações, tipo de construtor, métodos e atributos que uma classe Java deve conter para que se torne uma representação de determinada tabela do banco de dados da aplicação, estabelecendo assim um caminho bidirecional para a transformação de objetos em registros e registros em objetos.

Segundo *Oracle* [23], uma *Bean* deve possuir as seguintes características:

1) Apresentar a anotação `@Entity` acima da sua declaração; 2) Conter um construtor padrão sem argumentos; 3) Declarar apenas atributos privados que possam ter seus valores lidos e escritos; 4) Fornecer métodos `Getter` e `Setter` para manipulação desses atributos, não podendo apresentar outras operações que não sejam de leitura e escrita de dados.

Para gerar o código de classes *Bean* um subconjunto de itens do diagrama de classes e de suas propriedades são necessários, sendo denominados, nesta monografia, como itens de interesse. Tais itens fornecem a base para o processo de engenharia à frente implementado.

Uma breve descrição desses elementos e de suas representações na UML, em Java e no modelo objeto-relacional da JPA é dada nas próximas seções.

2.7 CLASSE

Classes são representações conceituais de itens que fazem parte do domínio do problema. Usadas para abstrair elementos (conceituais ou concretos) da aplicação, uma classe descreve um conjunto de instâncias (objetos) que compartilham as mesmas propriedades, comportamentos, relacionamentos e semântica [3]. Dentro de um modelo, a classe é identificada por um nome, que além de único, deve ter relação direta com o item ao qual ela abstrai [8]. Classes podem conter propriedades e métodos que definem sua estrutura e comportamento.

Em Java, classes são classificadores que dão origem a objetos. A nível objeto-relacional, elas representam entidades, cujos objetos são convertidos em registros. A Figura 8 mostra a representação da classe na UML.

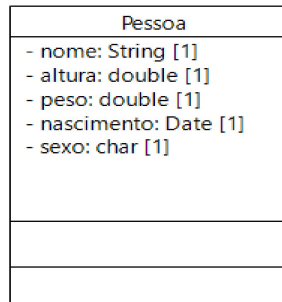


Figura 8 – Classe Pessoa.

2.7.1 Atributo da Classe

O atributo é uma abstração que apresenta determinada característica da classe, compartilhada por todos os seus objetos. Um atributo especifica um intervalo de valores para uma propriedade do classificador, que serve para caracterizar e diferenciar os objetos [8]. Na classe, o atributo é identificado por um nome. Além do nome, outras informações podem estar atreladas a este elemento, como visibilidade e tipo [3].

- **Tipo:** característica que determina a natureza dos dados armazenados pela propriedade, restringindo seus valores a um subconjunto no espaço (números, caracteres, valores lógicos, etc).
- **Visibilidade:** usado para indicar o nível de acesso à propriedade. A visibilidade representa um mecanismo de encapsulamento da informação. Existem quatro tipos distintos na UML e em Java: privado, público, protegido e pacote [8].

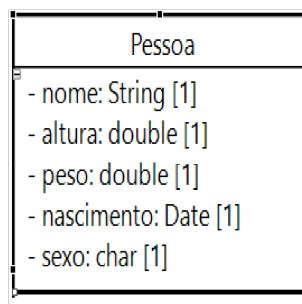


Figura 9 – Atributos Privados da Classe Pessoa.

A nível de implementação OO os atributos caracterizam a classe, definindo suas propriedades. Já a nível objeto-relacional, tais elementos dão origem à colunas nas entidades, onde são armazenados os dados dos registros. A Figura 9 mostra como os atributos se organizam na classe.

2.8 RELACIONAMENTOS

Relacionamentos são usados para descrever mecanismos de colaboração entre as classes e também na representação de conceitos mais ricos da orientação a objetos [3]. Um relacionamento indica uma conexão entre classes, descrevendo um caminho para a troca de informações ou um mecanismo de cooperação entre instâncias [8].

A UML apresenta três tipos básicos de relacionamento: associação, generalização e dependência. Apenas dois deles (associação e generalização) possuem relação direta com a implementação.

2.8.1 Generalização

O relacionamento de generalização trabalha com os conceitos de superclasse e subclasse na representação de heranças. Na generalização, classes mais gerais compartilham seus atributos e operações com outras que apresentem estas mesmas características, mas que ao mesmo tempo tenham diferenças pontuais de estrutura e comportamento [8]. As classes filhas (subclasses) herdam todos os atributos, operações e relacionamentos das suas classes mães (superclasses). A Figura 10 apresenta um relacionamento de generalização na UML.

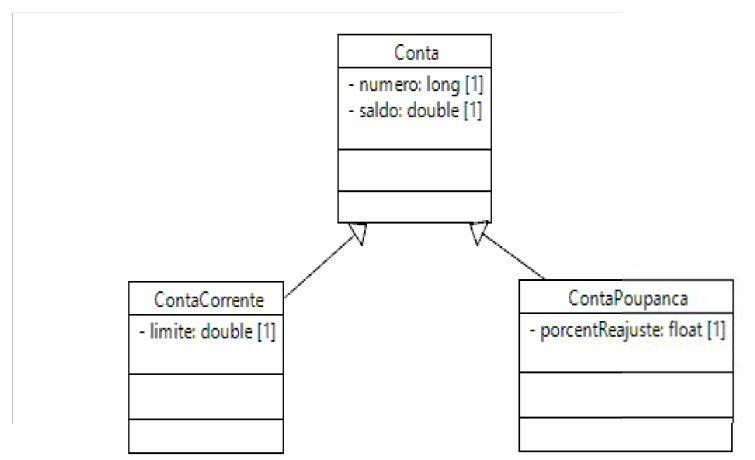


Figura 10 – Herança na UML.

Em Java, apenas heranças simples são suportadas. A nível objeto-relacional, generalizações são mapeadas no banco através de referências entre tabelas.

2.8.2 Associação

Uma associação representa um relacionamento estrutural entre itens, indicando que existe um caminho para troca de dados entre os objetos das classes envolvidas. Associações podem conectar uma, duas ou N-classes [8]. Dependendo da quantidade de classes conectadas, uma associação pode ser classificada em unária (Figura 12), binária (Figura 11) ou N-ária.

Associações binárias são as mais comuns no diagrama de classes, já as N-árias devem ter seu uso evitado por dificultarem a leitura do modelo e abrirem espaço para interpretações ambíguas [3]. A informação básica da associação é seu nome, o qual descreve sua função no modelo. Além do nome, uma associação também pode apresentar:

- **Multiplicidade:** determina a quantidade mínima e máxima de instâncias que pode estar conectadas a cada uma das extremidades da associação. Por padrão assume o valor 1..1 (Um para Um), podendo ser alterada para 0..* (Zero ou Muitos), 1..* (Um ou Muitos), * (Muitos). Por meio da multiplicidade é possível controlar a quantidade de objetos envolvidos na associação.
- **Papel:** usado como uma interface externa da classe. Geralmente relaciona-se com atributos da implementação.
- **Navegabilidade:** define o sentido em que os objetos podem trafegar na associação: $A < - > B$ (bidirecional), $A - > B$ (apenas A acessa B) ou $A < - B$ (apenas B acessa A).

A nível de código Java associações dão origem à atributos nas classes com extremidade navegáveis. Nesse caso, o papel define a implementação do atributo. Já a nível objeto-relacional as associações, assim como as generalizações, são representadas por meio de referências entre tabelas.



Figura 11 – Associação Binária Simples.

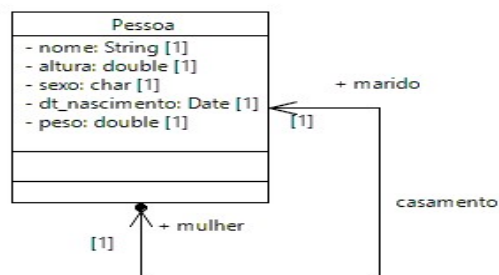


Figura 12 – Associação Unária ou Reflexiva.

2.8.3 Agregação e Composição

A agregação e a composição são associações que, ao contrário das simples, definem níveis de importância às classes envolvidas, trazendo significado extra ao modelo e consequentemente à implementação. São associações binárias, onde uma classe, que representa o elemento todo, possui relação de dependência ou de controle com outra classe, que contém suas partes [8]. A relação do todo com as partes determina o tipo de associação. A Figura 13 apresenta como agregações e composições são representadas na UML.

- **Agregação:** nesse tipo de associação a classe que representa o todo necessita ter suas informações complementadas pelos objetos parte [8]. A agregação indica que o todo só está completo quando em posse de todas as suas partes. Nesse tipo de associação, o todo é dependente das partes.
- **Composição:** variação da agregação. Na composição, invés do objeto todo ser dependente de suas partes, ocorre o contrário. Essa associação especifica que instâncias da classe todo possuem controle total de suas partes, sendo responsável pela criação e destruição desses objetos. A composição estabelece que uma parte não pode existir sem seu todo.

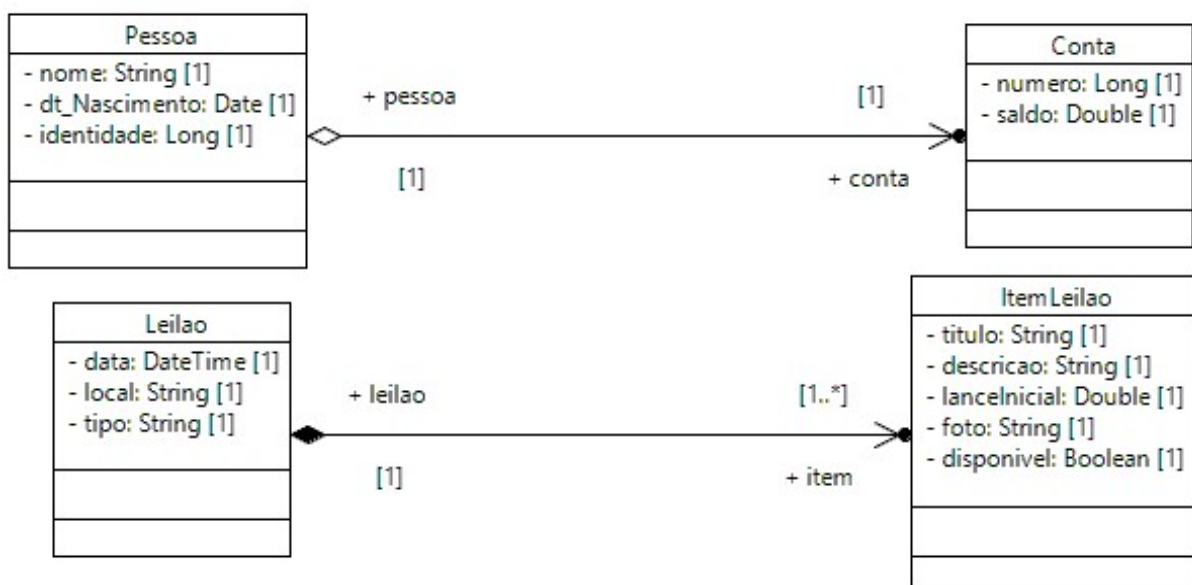


Figura 13 – Exemplos de Agregação e Composição.

3 TRABALHOS RELACIONADOS

Esta capítulo apresenta trabalhos cujo conteúdo descreve ferramentas de transformação que implementam a engenharia à frente a partir de diagramas de classes, tendo a linguagem Java como destino. São mostradas as principais funcionalidades de cada ferramenta, bem como as deficiências observadas em seu funcionamento.

Os problemas verificados serviram para tornar a ferramenta proposta mais completa.

3.1 *GENCODE*

A *GenCode* [14] é uma ferramenta de apoio ao desenvolvimento de sistemas embarcados, capaz de ler, capturar e gerar código Java a partir de diagramas de classes. Ela inclui a especificação de classe, classe abstrata, superclasse, especialização, pacotes, declaração de atributos, métodos e construtores, além de `gets` e `sets`.

O processo de transformação implementado na aplicação é composto de duas etapas: extração dos dados e geração do código. Na primeira, o diagrama de entrada em formato XMI tem seus dados lidos e armazenados na memória. Essa tarefa é executada utilizando a biblioteca de manipulação de arquivos do Java: `java.io`. Na segunda, tomando como referência a equivalência de conceitos da orientação a objetos, cada componente do diagrama de classes é mapeado de forma direta para seu elemento correspondente na linguagem de destino. Assim, ao final da transformação cada classe do modelo gera uma classe Java, propriedades dessas classes geram atributos e assim por diante.

A *GenCode*, apesar de robusta, apresenta alguns problemas.

- A leitura e extração dos dados do modelo são feitas exclusivamente utilizando a API nativa de arquivos do Java, não havendo qualquer apoio de motores de transformação ou bibliotecas de processamento XML. Essa implementação pode gerar problemas de desempenho durante o processamento de diagramas mais extensos, devido a complexidade do arquivo XMI.
- Nomes das classes, atributos e métodos, assim como todos os outros itens do modelo não passam por uma validação antes de serem transformados em código, o que pode resultar em erros de compilação, caso as notações destes elementos estejam fora dos padrões da linguagem Java.
- A ferramenta não conta com uma interface gráfica, dificultando a interação com o usuário.
- A navegabilidade é respeitada durante a transformação. Todavia, agregações e composições não são transcritas para o código.

3.2 HIBEROBJECTS

A *HiberObjects* [24] é uma ferramenta de modelagem e geração de código para diversas plataformas como *Java Persistence API*, classes *Hibernate* e serviços *Google Web Toolkit* (GWT). Com *Hiberobjects* é possível gerar o esqueleto do sistema através de diagramas de classes e o corpo dos métodos e construtores por meio de diagramas de sequência, sendo esse seu principal diferencial.

No módulo *Hibernate* da aplicação, as entidades persistentes recebem o estereótipo *Persistent* e as anotações de mapeamento são adicionadas ao diagrama por meio do elemento *Annotation* do diagrama de classes (Figura 14). A *Hiberobjects* funciona como um *plugin* do eclipse e possui um módulo de modelagem proprietário para criação de diagramas, sendo esse seu principal ponto negativo.

O módulo de modelagem fornece apenas um número reduzido de componentes do diagrama de classes, limitando as opções do projetista. Além disso, a sua interface deixa a desejar no quesito usabilidade quando comparada a outras ferramentas *CASE*. A Figura 14 apresenta um diagrama de classes modelado utilizando a ferramenta.

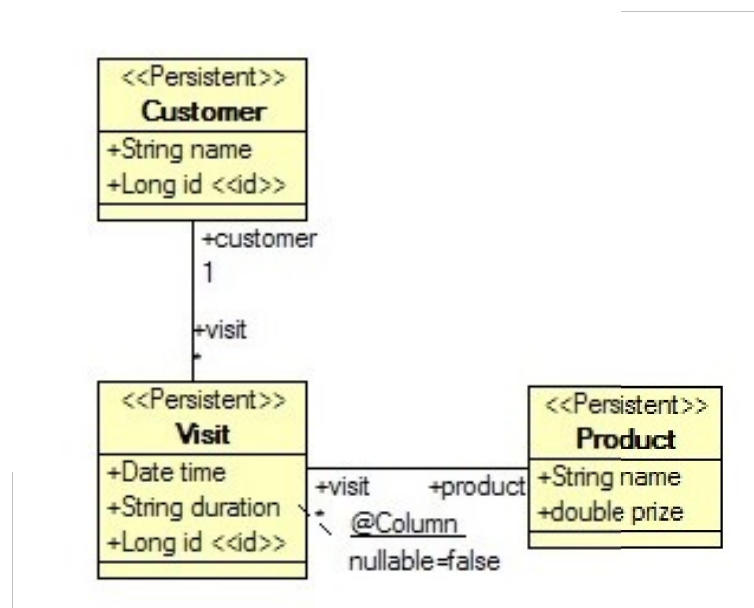


Figura 14 – Diagrama de Classes Modelado na *HiberObjects*. Adaptado de *Cgernert* [6].

3.3 MD-JPA

O *MD-JPA* [7] especifica um perfil UML para modelagem de diagramas de classes específicos para o padrão JPA e um módulo de transformação de código fonte para estes modelos. Esse perfil permite a representação dos aspectos estruturais de banco dados e sistemas de informação em uma abordagem dirigida por modelos através de mecanismos de extensão da UML (estereótipos). No *MD-JPA*, o metamodelo do diagrama de classes teve sua notação

estendida para representar dados das anotações da *Java Persistence API*. O perfil implementado na ferramenta trabalha com o conceito de enriquecimento de modelos da MDA, onde um diagrama simples, recebido como entrada, é detalhado até se tornar um PSM.

O objetivo da *MD-JPA* é permitir a modelagem completa da persistência relacional em conjunto com os conceitos da orientação a objetos. Como consequência, seus diagramas tendem a apresentar fortíssimo relacionamento com a API JPA. Para que o código seja gerado, os estereótipos do perfil devem ser atribuídos a nível de elemento do modelo de forma individual. Sendo assim, classes, heranças, atributos só podem ser transformados caso recebam um estereótipo que os mapeie para o código. Essa é uma das principais desvantagens do perfil, pois termina dificultando a tarefa do projetista, visto que mais tempo deverá ser despendido para o aprendizado da nova notação da UML e na construção dos modelos.

No processo de engenharia à frente, foram utilizados os conceitos de transformações entre metamodelos. O digrama de classes *MD-JPA* é inicialmente transformado em uma representação no metamodelo do Java e depois transportado para implementação da linguagem por meio de bibliotecas da IDE eclipse. O código gerado é uma abstração completa do modelo do banco de dados do sistema. No entanto, ainda ocorre a perda de informação, pois, segundo o desenvolvedor, optou-se por não atribuir nenhuma semântica adicional de agregações e composições, sendo elas tratadas como associação simples durante a geração do código.

Por fazer uso de bibliotecas da IDE eclipse a *MD-JPA* só pode ser usada dentro do ambiente, possuindo assim uma dependência. A Figura 15 mostra um modelo PSM da *MD-JPA*.

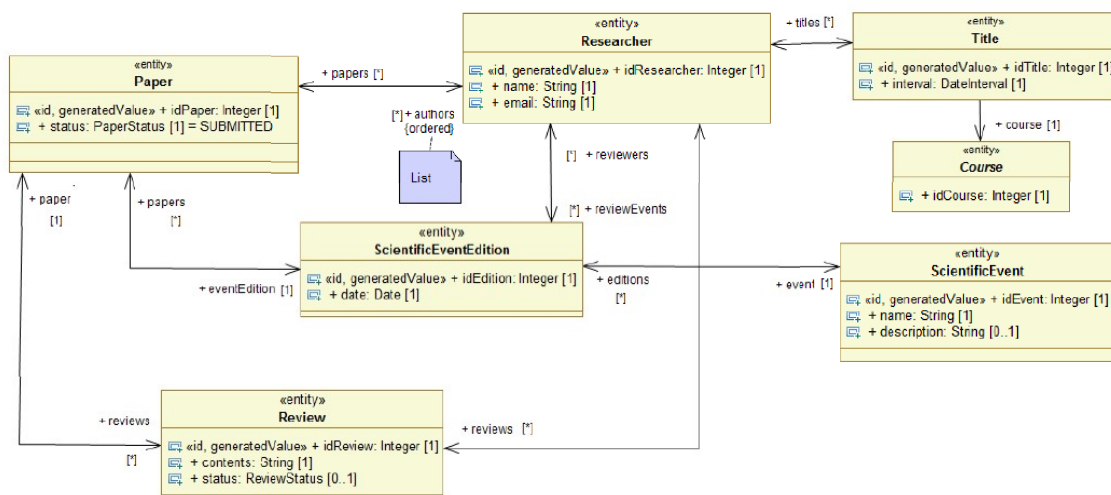


Figura 15 – Diagrama de Classes no padrão *MD-JPA* [7]

4 GENJPA

A construção de novas ferramentas de apoio às MDs, mais precisamente daquelas que realizam o processo de engenharia à frente, se faz necessária por duas razões: alguns domínios de aplicação ainda não possuem ferramentas do tipo ou porque as ferramentas existentes dentro de um domínio apresentam deficiências ao transformar modelo em código.

Grande parte das ferramentas de transformação, sejam elas desenvolvidas no meio acadêmico ou pela própria indústria, apresentam problemas de consistência entre modelo e código gerado. Muitas vezes, ao se depararem com conceitos mais complexos da UML, tais ferramentas terminam ignorando a existência desses elementos no diagrama, não mapeando-os durante a transformação para o código [1] [9].

Durante a análise de aplicações que realizam transformações do diagramas de classes para a linguagem Java, verificou-se que elementos semanticamente mais ricos do diagrama de classes nunca tinham seu significado transportado ao código, havendo assim uma perda de informações. Observou-se também a deficiência dessas ferramentas no quesito interação humano-computador: uma delas não apresentavam GUI, mecanismo básico para interação, já aquelas que apresentavam limitavam o projetista a trabalhar conforme suas próprias convenções, não oferecendo mecanismos para geração de código mais condizente com a realidade da aplicação desenvolvida.

Percebeu-se então a necessidade de uma ferramenta de apoio ao processo de engenharia à frente capaz de sanar ou minimizar a ocorrência destes problemas, isto é, uma ferramenta que diminuísse a ocorrência de inconsistências e que fornecesse mecanismos de interação ao usuário para o controle de algumas características do mapeamento. Seguindo esses preceitos foi concebida a *GenJPA*.

A *GenJPA* é uma ferramenta de transformação que trabalha com diagramas de classes da UML modelados conforme a perspectiva de dados, fazendo um mapeamento desses modelos para código Java enriquecido com anotações JPA. Como resultado, a ferramenta é capaz de retornar uma implementação das classes persistentes do *softwares* (Entidades), que trabalham a nível objeto-relacional intermediando a troca de informações entre aplicação e banco de dados.

No decorrer do desenvolvimento da aplicação, além dos fatores associados ao mapeamento, também levou-se em consideração aspectos relacionados à consistência da transformação modelo-código e da interação com o usuário. Buscaram-se maneiras de diminuir a perda de informações durante o processo de engenharia à frente e de tornar o projetista membro partícipe da transformação.

Para isso, regras especiais de mapeamento e uma interface gráfica com o usuário foram desenvolvidas. Por meio dessas regras, baseadas em anotações e elementos da JPA, conceitos

como o de agregação e composição puderam ser devidamente transportados para o código. A interface apresenta-se como elemento intermediador, oferecendo ao projetista a possibilidade de repassar informações que serão usadas pela ferramenta durante a geração do código, como estruturas de dados e mapeamento de heranças.

O processo de engenharia à frente implementado na *GenJPA* é realizado em três etapas: extração das informações do diagrama, validação dos dados e geração do código. Cada uma destas etapas é executada por um módulo específico que utiliza uma ou mais tecnologias da MDA. Todos os módulos se encontram integrados em uma mesma aplicação.

A *GenJPA* foi desenvolvida em Java e executa em ambiente *Desktop* na forma de um formulário *SWING* (Figura 16). Na leitura do diagrama, optou-se pela utilização de modelos no formato XMI por ser um padrão para troca de diagramas entre ferramentas *CASE* e de transformação. Com isso foi possível desvincular a ferramenta implementada da ferramenta *CASE* usada na modelagem dos diagramas.

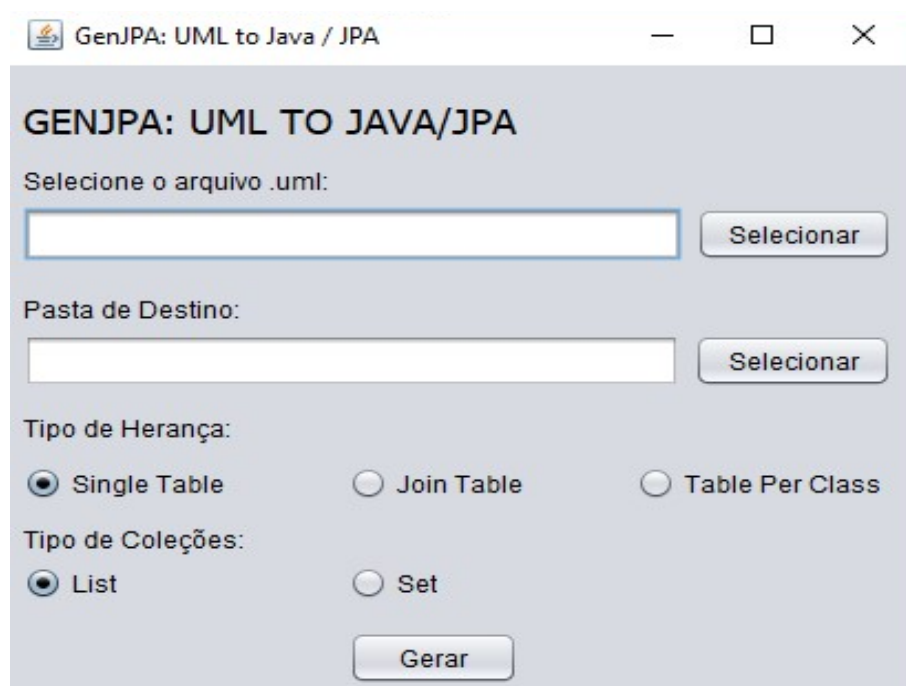


Figura 16 – Interface da *GenJPA*.

O código resultante do processo de engenharia à frente pode ser utilizadas em qualquer sistema que faça uso de banco de dados relacionais. Sendo assim, sistemas *Web*, *Desktop* e até mesmo *Mobile* podem ter seu tempo de implementação diminuído com a ajuda da *GenJPA*.

As próximas seções descrevem o processo de engenharia à frente implementado na aplicação. São apresentados os componentes, tecnologias e funcionalidades de cada módulo da ferramenta.

No decorrer de cada seção, alguns diagramas são usados para representar os conceitos

apresentados. Todos esses diagramas foram modelados utilizando o Papyrus [26], um *plugin* do Eclipse que fornece suporte à representação de diagramas no formato XMI. A Figura 17 apresenta uma visão geral do método de transformação desenvolvido.

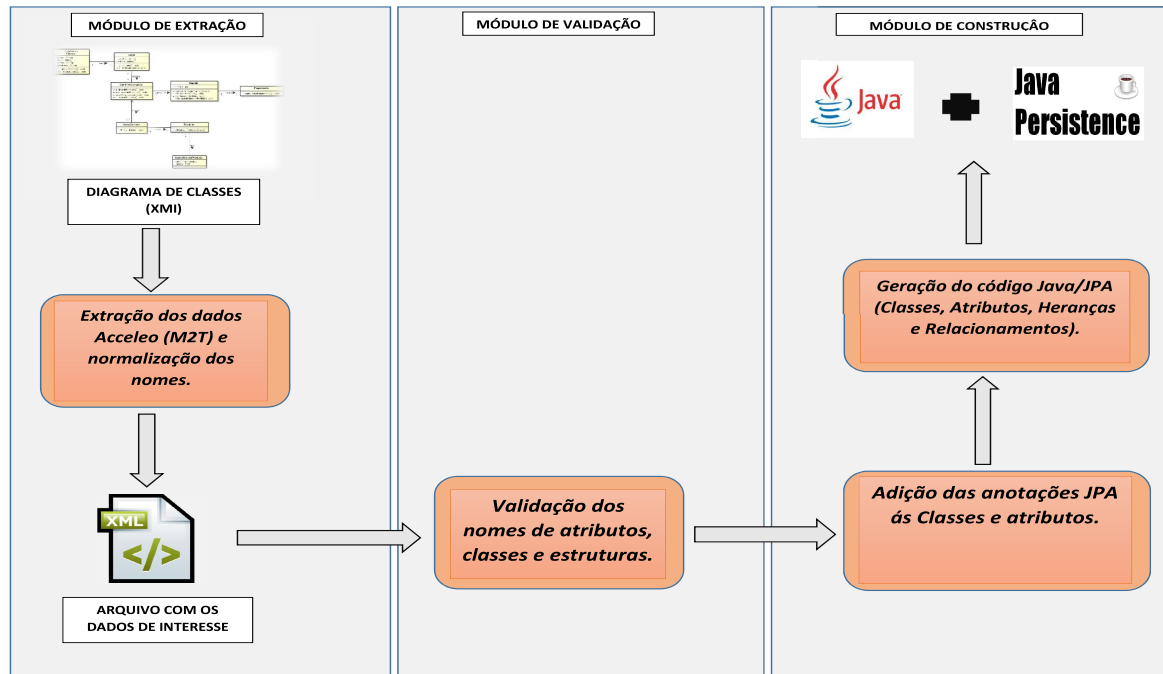


Figura 17 – Funcionamento da *GenJPA*.

4.1 MÓDULO DE EXTRAÇÃO

O módulo de extração percorre o modelo de entrada, recuperando as informações dos itens de interesse do diagrama e estruturando esses dados para um formato que facilite sua recuperação pelos módulos de validação e mapeamento da aplicação. Esse procedimento é realizado através de transformações do tipo modelo-texto, onde instâncias de elementos da UML, inicialmente estruturadas a nível de metamodelo no formato XMI, são convertidas em *tags* inseridas em um arquivo XML. Como resultado desse processo, obtêm-se um arquivo bem formado contendo apenas as informações necessárias para a geração do código. Na implementação dessa funcionalidade, utilizou-se a Linguagem de Transformação Modelo-Texto *Acceleio*.

Para cada bloco do diagrama de classes descrito, uma regra de transformação em *Acceleio* foi implementada. Essas regras seguem a mesma definição: Selecione todas as instâncias de determinado bloco do modelo, extraia suas propriedades de interesse e escreva esses dados no arquivo de saída em formato estruturado.

A etapa de extração é finalizada quando todas as regras tiverem sido executadas, isto é,

quando todos os dados de interesse haverem sido escritos no arquivo XML de saída. As Tabelas 1 e 2 apresentam todas as tags geradas pelas regras de transformação *Acceleo*.

Item do Diagrama	Elemento do Metamodelo	Tag Gerada na Saída
Classe	<i>Class</i>	<classe></classe>
Atributo	<i>Property</i>	<atributo></atributo>
Associação	<i>Association</i>	<associacao></associacao>
Generalização	<i>Generalization</i>	<heranca></heranca>

Tabela 1 – Itens Básicos do Diagrama de Classes.

Item	Propriedade	Descrição	Tag Gerada
<i>Class</i>	<i>name</i>	Nome da Classe	<id></id> <nome></nome>
<i>Property</i>	<i>name</i>	Nome do Atributo	<nome></nome>
	<i>type</i>	Tipo do Atributo	<tipo></tipo>
	<i>visibility</i>	Visibilidade do Atributo	<visibilidade></visibilidade>
	<i>default</i>	Valor Default	<default></default>
<i>Generalization</i>	<i>general</i>	Superclasse da Herança	<general></general>
	<i>specific</i>	Subclasse da Herança	<specific></specific>
<i>Association</i>	<i>name</i>	Identificador da Associação	<nome></nome>
<i>Extremidade A</i>	<i>type</i>	Classe da Extremidade A	<classe></classe>
	<i>agregation</i>	Tipo de agregação lado A	<agregacao></agregacao>
	<i>lower</i>	Valor mínimo da multiplicidade A	<minimo></minimo>
	<i>upper</i>	Valor máximo da multiplicidade A	<maximo></maximo>
	<i>isNavigable</i>	Identifica se o lado A é navegável	<navegavel></navegavel>
<i>Extremidade B</i>	<i>type</i>	Classe da Extremidade B	<classe></classe>
	<i>agregation</i>	Tipo de agregação lado B	<agregacao></agregacao>
	<i>lower</i>	Valor mínimo da multiplicidade B	<minimo></minimo>
	<i>upper</i>	Valor máximo da multiplicidade B	<maximo></maximo>
	<i>isNavigable</i>	Identifica se o lado B é navegável	<navegavel></navegavel>

Tabela 2 – Propriedades Específicas de Cada Item.

4.1.1 Normalização de Nomes

Durante a extração dos dados, algumas propriedades (nomes de classes e atributos) passam por processamento adicional com o objetivo de padronizá-los às recomendações da linguagem Java. A classe `NormalizadorDeNomes.java` do módulo de extração fornece métodos que implementam essas convenções, formatando o conteúdo das propriedades nomeáveis antes de serem transformados em tags no arquivo de saída.

Nesse processo, espaços em branco, caracteres especiais e acentos são removidos do texto. Além da remoção desses caracteres, a classe executa formatações adicionais no texto para que fique de acordo com as seguintes convenções do Java:

1) Nomes de classes devem iniciar com letra maiúscula; 2) Nomes de atributos devem iniciar com letra minúscula; 3) Nomes compostos devem possuir a primeira letra de cada palavra após a primeira em maiúsculo; 4) Nomes de classes e atributos só podem ser compostos por letras, números e pelo caractere especial `_`; 5) Recomenda-se que os nomes contenham apenas caracteres suportado pelo padrão ASCII;

4.1.2 Resultados da Etapa de Extração

Ao final da fase de extração, o arquivo `model.xml` é gerado. Esse arquivo contém todas as informações necessárias para geração das entidades e de seus relacionamentos, sendo usado pelo módulo de validação na verificação do diagrama.

4.2 FASE DE VALIDAÇÃO

Na etapa de validação, os dados contidos no arquivo `model.xml` são usados na instanciação das classes do pacote `modelo` da *GenJPA*. Estas classes são abstrações de itens do Java e da UML usados pelos módulos de validação e construção na execução de suas funções. O módulo de verificação trabalha como intermediador entre diagrama e código, transformando elementos do modelo em representações mais próximas do resultado final, bem como realizando a validação desses itens.

Nessa etapa, a ferramenta faz uma verificação minuciosa do conteúdo e estrutura dos objetos gerados a partir da leitura do modelo, visando a detecção de construções incompatíveis com a linguagem Java. Todos os itens de interesse são validadas individualmente, tendo seus erros agrupados em listas. Nesse processo, são identificados elementos cuja notação causariam erros de compilação quando transcritos para o código.

A definição das regras de validações se deu em observância à sintaxe da linguagem Java e de suas limitações, bem como considerando algumas das convenções da *Java Persistence API* relacionadas aos tipos de dados de classes *Bean*.

4.2.1 Interpretação do Arquivo XML

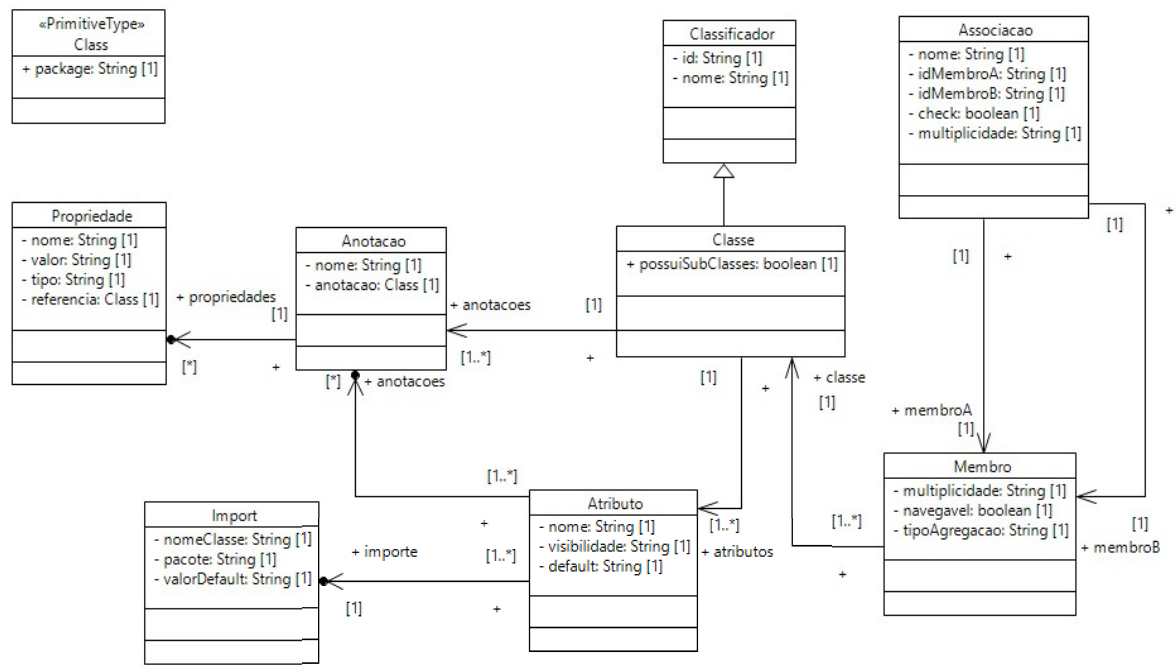
Utilizando a API JDOM [27] a classe `CarregaArquivoXML.java` (Figura 18) faz a leitura das tags contidas no arquivo `model`, agrupando-as em torno do bloco do diagrama que as define. Para cada item do modelo, foi implementada uma classe (Figura 19) cujos atributos abstraem a notação do elemento no arquivo XML, permitindo que os dados contidos nas tags possam ser utilizados na instanciização de seus objetos.

A classe `CarregaArquivoXML.java` (Figura 18) faz uso dessas itens ao transportar as instâncias das classes, atributos e associações para a aplicação. Ela lê os dados contidos nas tags que abstraem esses elementos e os utiliza na criação dos objetos correspondentes. Essas instâncias são então agrupadas em listas para que possam ser recuperadas de maneira mais ágil pela aplicação.

CarregaArquivoXML
<pre>- classes: Map<String, Classe> [1] - associacoes: List<Associacao> [1] - herancas: Map<String, String> [1]</pre>
<pre>- carregaClasses() - gerarAssociacoes() + getClasses(): Map<String, Classe> + getHerancas(): Map<String, String> + getAssociacoes(): List<Associacao></pre>

Figura 18 – Classe `CarregaArquivoXML.java`

Ao final da leitura, três listas são geradas dentro da classe `CarregaArquivoXML`: uma contendo as instâncias das classes, outra com as associações e a última com as heranças. Essas listas estáticas permanecem na memória da aplicação até o fim da sua execução, sendo usadas no processo de validação do modelo e construção do código.

Figura 19 – Classes de Modelo da *GenJPA*.

4.2.2 Validação dos Objetos

O módulo de validação da *GenJPA* implementa regras que verificam a integridade dos blocos do modelo, identificando aqueles cuja notação apresenta alguma incompatibilidade com o Java ou que ferem construções da UML. Esse módulo processa as listas de classes, associações e heranças, sinalizando, por meio de exceções, elementos inválidos do modelo.

Na ferramenta, quatro tipos de *Exception* foram implementadas para execução dessa sinalização. Definidas como classes Java, cada uma identifica a existência de erros na definição do elemento ou de suas propriedades. A Tabela 3 descreve essas exceções e a situação em que são reportadas.

Exceção Java	Ocorrência
<code>NomeInvalidoException.java</code>	Reportada quando o módulo de validação identifica erros de nomenclatura nas classes e atributos.
<code>TipoInvalidoException.java</code>	Reportada quando o módulo de validação identifica erros de tipagem nos atributos das classes.
<code>HerancaInvalidaException.java</code>	Reportada quando o módulo de validação identifica generalizações que não estejam de acordo com a sintaxe Java.
<code>AssociacaoInvalidaException.java</code>	Reportada quando o módulo de validação identifica associações em desacordo com as especificações da UML.

Tabela 3 – Exceções da *GenJPA*.

Durante a verificação, diversas exceções podem ser retornadas pelo módulo de validação. Cada ocorrência identifica um elemento inválido do diagrama, o qual deverá passar por reestruturação antes de ser transformado em código. A *GenJPA* agrupa essas ocorrências de modo que, ao final da validação, possam ser usadas para guiar as alterações necessárias no modelo. Para isso, todas as exceções, assim que lançadas, tem seus dados capturadas e estruturadas como objetos da classe `Erro.java` (Figura 20): tal erro, além de identificar seu elemento causador, fornece ainda uma descrição detalhada do problema ocorrido.

Incorporada ao módulo de validação, uma lista recebe todas as instância de `Erro.java`. A mesma é usada ao final da etapa para geração do arquivo de `Log` da aplicação, cujo conteúdo identifica todos os elementos que impedem o prosseguimento da transformação.

Erro
- elemento: String [1] - tipo: String [1] - mensagem: String [1]

Figura 20 – `Erro.java`.

4.2.2.1 Validação dos Nomes

Mesmo depois de normalizados nomes de classes e atributos podem conter sequência de caracteres cujo conteúdo gerariam erros de compilação quando transformados em código executável. Exceções do tipo `NomeInvalidoException.java` são lançadas pela ferramenta sempre que algum nome descumpra as seguintes convenções:

- Nomes de classes, atributos ou de qualquer outro item do código não podem conter palavras reservadas da linguagem Java:

```
"abstract", "assert", "boolean", "break", "byte", "case", "catch", "if",
"char", "class", "const", "continue", "default", "do", "double", "id"
"else", "enum", "extends", "false", "final", "finally", "float", "for",
"goto", "this", "implements", "import", "instanceof", "int", "interface",
"long", "native", "new", "null", "package", "private", "protected", "public",
"return", "short", "static", "strictfp", "super", "switch", "synchronized",
"throw", "throws", "transient", "true", "try", "void", "volatile", "while"
```

- Nomes de classes e atributos devem ser iniciados por uma letra ou pelo caractere \$ (cifrão). Números e o caractere especial _ (underline) podem ser usados no corpo dos nomes, mas nunca iniciando-os;

4.2.2.2 Validação dos Tipos

Na *GenJPA*, existe uma classe cujas funções são realizar o controle das importações, a conversão e a validação dos tipos de atributos. Essa classe (`GerenciadorDeTipos.java`) mantém um conjunto de listas contendo referências para os tipos primitivos, *Wrappers*, temporais, listas, conjuntos e mapas do Java; e também para tipos da UML e proprietários das principais ferramentas de modelagem.

Em posse desses dados, a (`GerenciadorDeTipos.java`) faz as validações referentes aos tipos de atributos, verificando se os tipos contidos no modelo possuem correspondentes em Java. Para retornar essa informação, a classe percorre todos atributos do diagrama verificando se os valores contidos em suas propriedades "tipo" estão referenciando valores da UML ou do Java. Nesse processo, atributos que tenham tipos relacionados à UML têm seus valores alterados para Java.

Para isso, a classe faz uma pesquisa em sua lista de referências da linguagem de destino, buscando um tipo correspondente ao valor definido no atributo processado. Caso a busca tenha sucesso, o valor da propriedade "tipo" do objeto é alterado para o aquele encontrado na busca. Se esse valor representar um tipo externo ao pacote raiz do Java, uma instância de `Importe` é gerada pela classe de gerenciamento e relacionada ao atributo, indicando a necessidade de importação.

Exceções `TipoInvalidoException.java` são lançadas sempre que o tipo de um atributo do modelo não possua valor correspondente nas listas da classe `GerenciadorDeTipos.java`.

4.2.2.3 Validação das Generalizações

A linguagem Java suporta apenas heranças do tipo simples, isto é, uma subclasse pode ter apenas uma classe mãe. Por outro lado, a UML permite, no diagrama de classes, a modelagem de heranças múltiplas. Por conta dessa divergência, as generalizações contidas no modelo não podem ser diretamente mapeadas para o código.

Na *GenJPA*, heranças múltiplas são sinalizadas com `HerancaInvalidaException.java`.

4.2.2.4 Validação das Associações

As regras de verificação das associações foram implementadas embasadas no documento oficial de especificação da UML: o *UML Superstructure* [2]. Nesse documento, existem duas recomendações cuja definição influenciam diretamente na consistência e qualidade do código gerado. Exceções do tipo `AssociacaoInvalidaException.java` são lançadas pela ferramenta caso alguma das seguintes restrições seja descumprida:

- Ao menos uma das extremidades da associação deve ser navegável.
- Para que o todo mantenha o controle sobre suas partes, associações do tipo composição devem apresentar multiplicidade 1..1 para * (um para muitos).

4.2.3 Finalização das Validações

Ao final das verificações, o módulo de validação faz uma consulta à lista de erros para determinar o prosseguimento ou não do processo de engenharia à frente. Dependendo do resultado desta consulta, duas são as situações possíveis:

Quando a lista contém erros, o módulo transfere os dados da mesma para o arquivo de Log da ferramenta (Figura 22). Feito isso, uma mensagem é apresentada na interface da aplicação (Figura 21), informando ao usuário que o diagrama contém elementos cuja notação é incompatível com as transformações, devendo passar assim por reformulação. O processo é dado como finalizado e a ferramenta encerra sua execução.

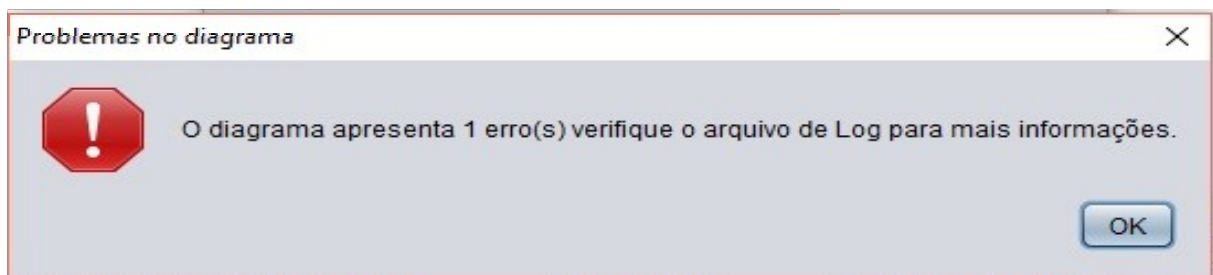


Figura 21 – *Feedback* ao Usuário.

```
<?xml version="1.0" encoding="UTF-8"?>
<erros>
  <Erro>
    <elemento>Conta</elemento>
    <tipo>Property</tipo>
    <descricaoErro>id : atributo com nome incompatível com a
      linguagem Java ou com o nome id.</descricaoErro>
  </Erro>
  <Erro>
    <elemento>Pessoa</elemento>
    <tipo>Property</tipo>
    <descricaoErro>id : atributo com nome incompatível com a
      linguagem Java ou com o nome id.</descricaoErro>
  </Erro>
</erros>
```

Figura 22 – Arquivo de Log.

Quando a lista se encontra vazia, o módulo de construção recebe uma mensagem informando que os blocos do modelo encontram-se em total acordo com as convenções da linguagem de destino, podendo assim passar pela transformação.

Os objetos contidos nas listas de classes, associações e heranças são então repassados a este módulo para que se encarregue de gerar o código final.

4.3 FASE DE CONSTRUÇÃO

Na etapa de construção, os objetos contidos nas listas de classes, associações e heranças são mapeadas para o código. Esse processo é executado por classes de controle, cuja função é transportar determinado conceito da UML para representações em código Java/JPA. Nessas classes, encontram-se definidas as construções sintáticas da linguagem Java, bem como da API JPA. Os blocos básico do modelo, assim como conceitos especiais da associação e tipos temporais, têm sua notação transcrita ao código durante a etapa.

A geração do código se dá em duas fases: mapeamento objeto-relacional / geração do código. Na primeira, os blocos recebem as propriedades da JPA na forma de anotações. Na segunda, as classes são agrupadas e passam pela transformação final, dando origem ao código.

Ao fim dessa fase, todas as entidades do sistema têm sua implementação gerada, podendo então ser usadas na criação do banco de dados da aplicação modelada.

4.3.1 Módulo de Controle

O módulo de controle agrupa todas as classes da *GenJPA* que mapeiam a linguagem de destino. Nesse módulo, as classes encontram-se divididas em dois grupos: as relacionadas com o Java e as conectadas com a JPA.

As classes de controle Java contém regras para geração da estrutura básica de uma classe na linguagem, fornecendo a sintaxe que define os conceitos de classe, atributos, heranças e métodos de controle de acesso (*Get* e *Set*). Já as classes de controle JPA abstraem as características do banco de dados, fornecendo a definição da sintaxe de anotações e propriedades da JPA, usadas no mapeamento objeto-relacional.

A Tabela 4 apresenta as classes de controle e suas funções.

Tipo de Código	Classe de Controle	Definição
Java	ClasseControlador.java	Contém a sintaxe dos conceitos de Importe, Construtor, Classe e Herança.
	AtributoControlador.java	Contém a sintaxe básica para definição de atributos, <i>gets</i> e <i>sets</i> .
	AssociaçãoControlador.java	Contém a sintaxe para a transformação de associações do modelo em atributos.
JPA	EntityControlador.java	Contém a sintaxe para definição de uma entidade.
	IdControlador.java	Contém a sintaxe para definição de chaves primárias em entidades.
	InheritanceControlador.java	Contém a sintaxe para mapeamento de heranças.
	TemporalControlador.java	Contém a sintaxe para mapeamento de tipos temporais.
	CascadeFetchControlador.java	Contém a sintaxe para mapeamento de agregações e composições.
	OneToOneControlador.java	Contém a sintaxe para mapeamento de associações 1..1.
	OneToManyManyToManyControlador.java	Contém a sintaxe para mapeamento de associações 1..*.
	ManyToManyControlador.java	Contém a sintaxe para mapeamento de associações *.*.

Tabela 4 – Classes de Controle.

4.4 MAPEAMENTO OBJETO-RELACIONAL

Nessa etapa, os objetos têm sua notação estendidas com anotações JPA para que possam ser transportados para o banco. Esse processo consiste na adição de anotações aos atributos e classes do modelo. Agregações, composições e associações simples são mapeadas para o código por esses elementos.

Para efeito explicativo, toma-se como base o diagrama da Figura 23. Todo o processo de mapeamento será demonstrado utilizando-o.

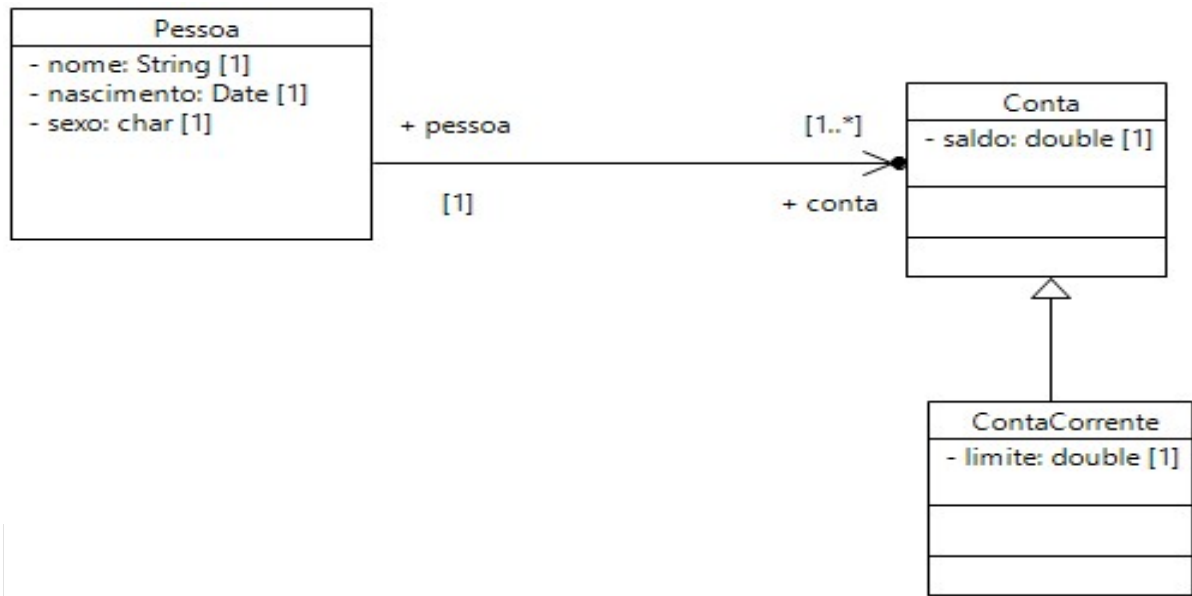


Figura 23 – Diagrama de Exemplo.

A Tabela 5 mostra quais as anotações JPA suportadas pela ferramenta e as classes de controle que as mapeiam. As subseções seguintes descrevem todo o processo de mapeamento objeto-relacional.

Classe de Mapeamento	Anotação da JPA
EntityControlador.java	@Entity
IdControlador.java	@Id
	@GeneratedValue
TemporalControlador.java	@Temporal
InheritanceControlador.java	@Inheritance
OneToOneControlador.java	@OneToOne
OneToManyManyToManyControlador.java	@OneToMany
	@ManyToOne
ManyToManyControlador.java	@ManyToMany

Tabela 5 – Anotações Suportadas pela *GenJPA*.

4.4.1 Transformação de Classes em Entidades

As classes passam por um mapeamento que indica sua natureza persistente. Três anotações JPA fornecidas pelas classes `EntityControlador.java` e `IdControlador.java` transformam classes comuns em entidades. Tais anotações são acopladas aos objetos do tipo classe tornando suas instâncias persistentes.

Aqui a propriedade `id` é gerada e adicionada como um atributo da classe. Essa propriedade serve como chave primária da classe no banco de dados.

- `@Entity`: identifica que a classe é uma entidade.
- `@Id`: acoplado ao atributo id identifica a chave primária da entidade.
- `@GeneratedValue`: acoplado ao atributo id indica que a chave primária deve ser gerada automaticamente pelo banco.

A Figura 24 mostra o diagrama de exemplo após passar pelo mapeamento.

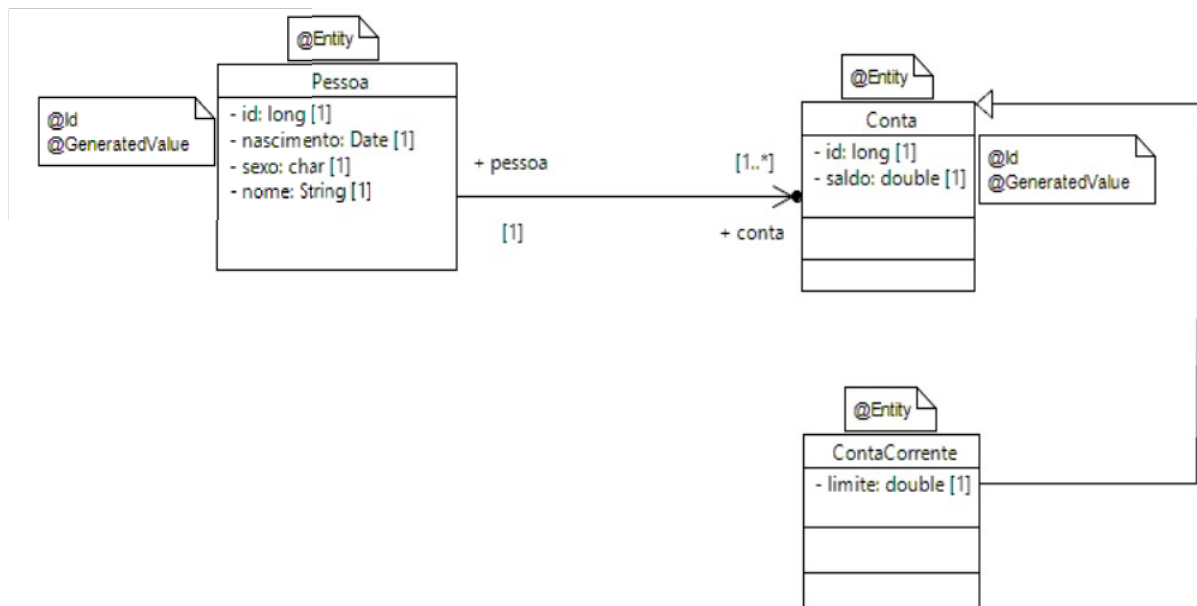


Figura 24 – Diagrama de Exemplo após a Transformação das Classes.

Todas as classes receberam a anotação `@Entity`. Um id foi gerado para a classe Pessoa e Conta. ContaCorrente não recebeu esta propriedade, pois herda do relacionamento de generalização.

4.4.2 Mapeamento de Heranças

As heranças do modelo são mapeadas para o banco pela classe `InheritanceControlador.java`. Ela abstrai a anotação `@Inheritance` da JPA, que durante o enriquecimento é atribuída a todas as superclasses da lista de classes.

O conceito de herança é mapeado de três maneiras distintas no banco relacional. Na interface da *GenJPA*, o usuário pode selecionar qual o tipo será usado durante a transformação.

- *Single Table*: Uma tabela única será criada contendo todos os atributos das classes participantes da herança.
- *Join Table*: Uma tabela é criada para cada classe mantendo a referência da herança.
- *Table per Class*: Uma tabela é criada para cada classe, todavia perdendo as referências da herança.

A Figura 25 mostra o diagrama de classes de exemplo após o mapeamento das generalizações.

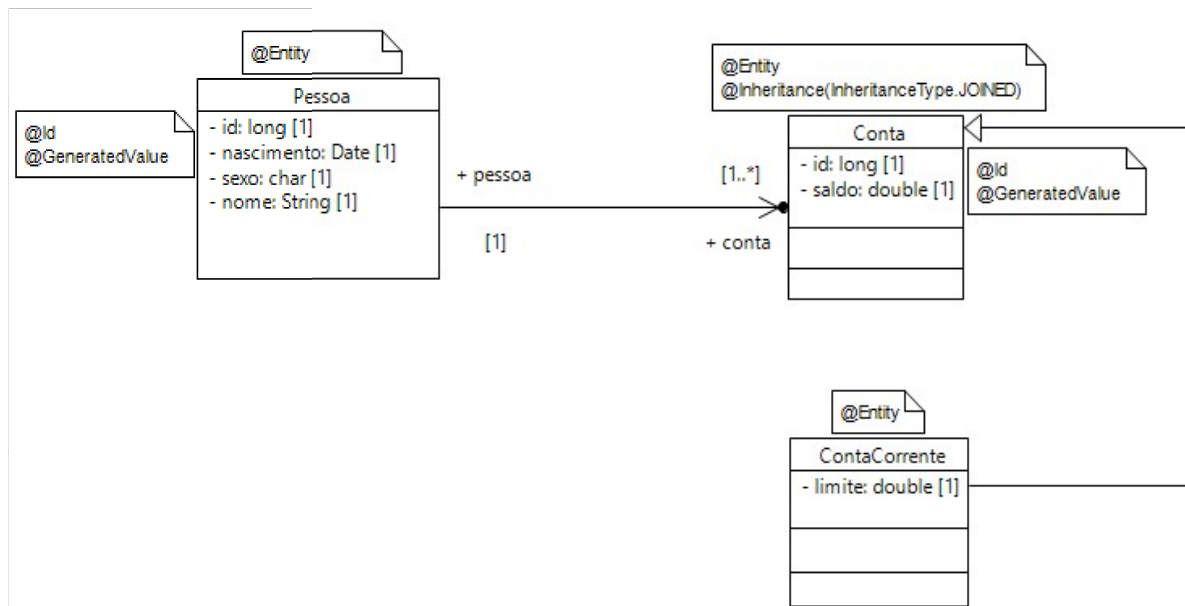


Figura 25 – Diagrama de Exemplo Após o Mapeamento das Generalizações.

A classe `Conta` recebeu a anotação `@Inheritance` do tipo `Join Table`. Esse é o valor padrão de mapeamento de generalizações caso o usuário não altere a opção na interface gráfica da ferramenta.

4.4.3 Mapeamento de Atributos Temporais

Atributos cujo tipo seja `java.util.Date` ou `java.util.Calendar` recebem a anotação `@Temporal` da JPA que indica ao banco a natureza temporal desses elementos.

A Figura 26 mostra o diagrama de exemplo após esse mapeamento.

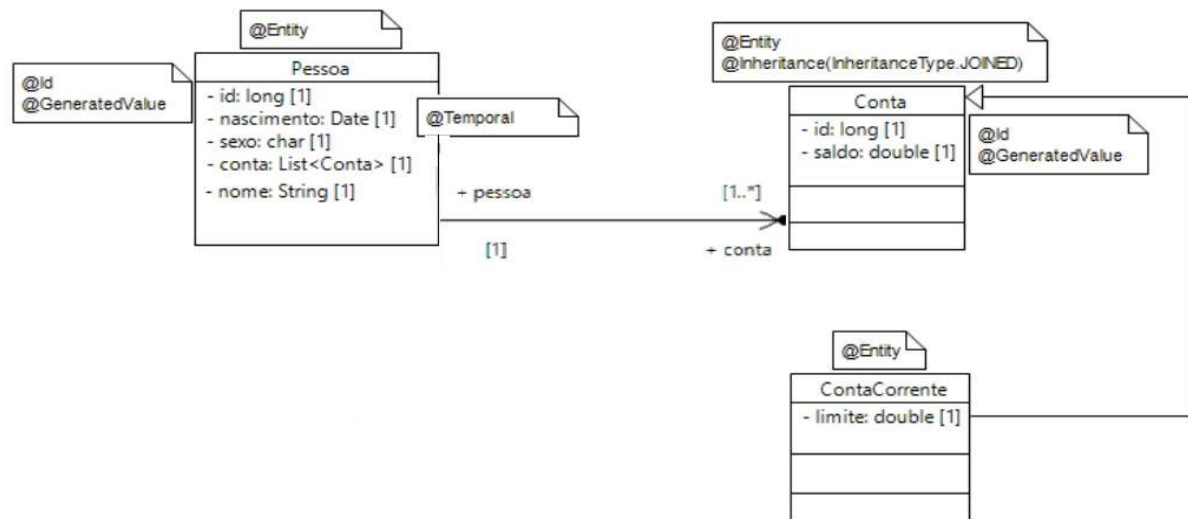


Figura 26 – Diagrama de Exemplo Após o Mapeamento dos Atributos Temporais.

O atributo nascimento da classe Pessoa recebeu a anotação `@Temporal`.

4.4.4 Mapeamento de Associações

As associações do modelo são mapeadas para o banco levando em consideração sua navegabilidade, multiplicidade e tipo de agregação. A JPA fornece quatro anotações para mapeamento de associações:

- `@OneToOne`: aplicado aos atributos gerados por associações cuja multiplicidade é 1..1 para 1..1 (Um para Um);
- `@ManyToOne`: aplicado aos atributos gerados por associações cuja multiplicidade é 1..* para 1..1 (Muitos para Um);
- `@OneToMany`: aplicado aos atributos gerados por associações cuja multiplicidade é 1..1 para 1..* (Um para Muitos);
- `@ManyToMany`: aplicado aos atributos gerados por associações cuja multiplicidade é * para * (Muitos para Muitos);

Para gerar o atributo correspondente à uma associação, a *GenJPA* verifica as multiplicidades de ambas as extremidades da mesma, bem como as navegabilidades. Após determinar a notação do atributo e criá-lo, a ferramenta acopla a anotação de mapeamento correspondente à propriedade.

A Figura 27 apresenta o diagrama de exemplo após ter sua associação mapeada.

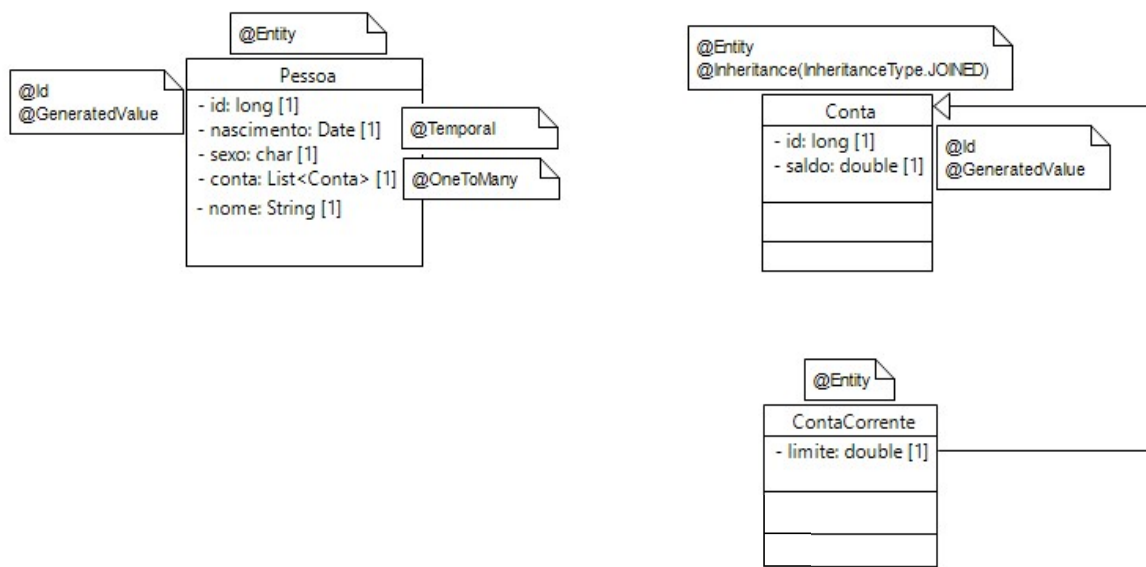


Figura 27 – Diagrama de Exemplo Após o Mapeamento das Associações.

A associação é transformada no atributo `conta` da classe `Pessoa`, o qual também recebeu a anotação `@OneToMany`. A navegabilidade da associação foi respeitada, mantendo-se as restrições do projeto.

4.4.4.1 Mapeamento de Composição

A propriedade *cascade* das anotações de mapeamento de associações indica a relação de dependência entre as classes. Essa propriedade determina como os registros devem ser tratados no banco. Se uma associação no diagrama é do tipo composição, a anotação de mapeamento do atributo resultante recebe a propriedade `cascade=CascadeType.ALL`. Ela indica a relação de controle entre a tabela todo e suas partes. A Figura 28 apresenta o resultado dessa etapa no diagrama de exemplo caso sua associação fosse do tipo composição.

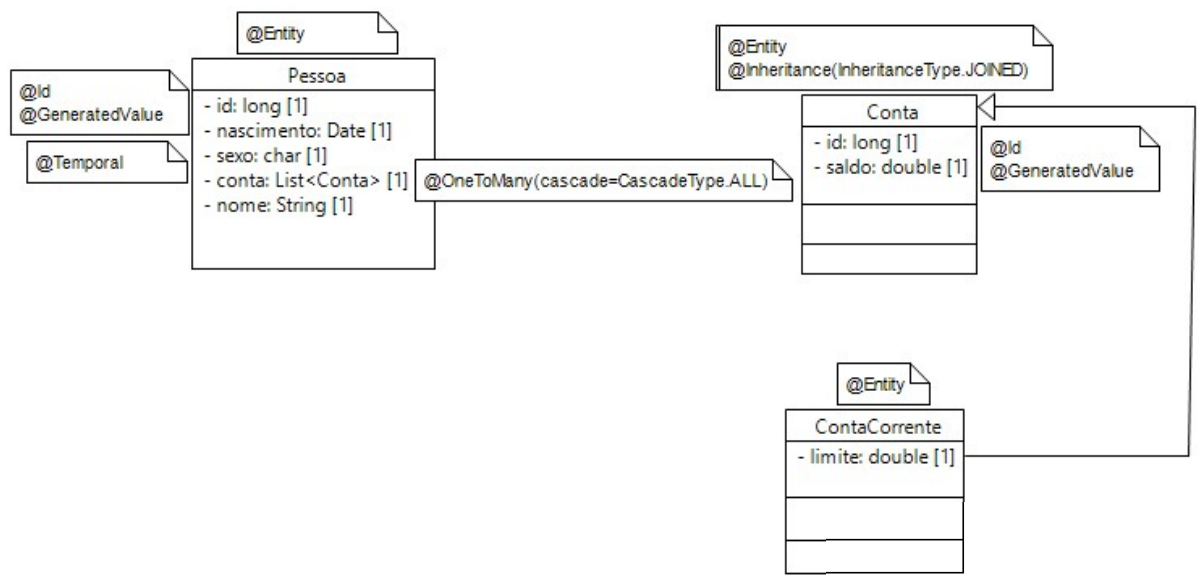


Figura 28 – Diagrama de Exemplo com Composição.

O atributo `conta` gerado pela associação recebe a anotação `@OneToMany` em conjunto com a propriedade `cascade=CascadeType.ALL`. Isso indica ao banco que todas as operações realizadas em **Pessoa** devem ser refletidas em seus objetos **conta**. Inclusões, exclusões e alterações feitas em **Pessoa** são imediatamente refletidas nas **Contas** relacionadas. Esse mapeamento mantém o significado da composição.

4.4.4.2 Mapeamento de Agregação

Para representar o conceito de agregação é utilizada a propriedade `fetch=FetchType.EAGER` no atributo gerado pela associação. Ela determina que as informações contidas nos registros sejam retornadas no momento em que o objeto seja consultado. A Figura 29 apresenta o resultado dessa etapa no diagrama de exemplo caso sua associação fosse do tipo agregação.

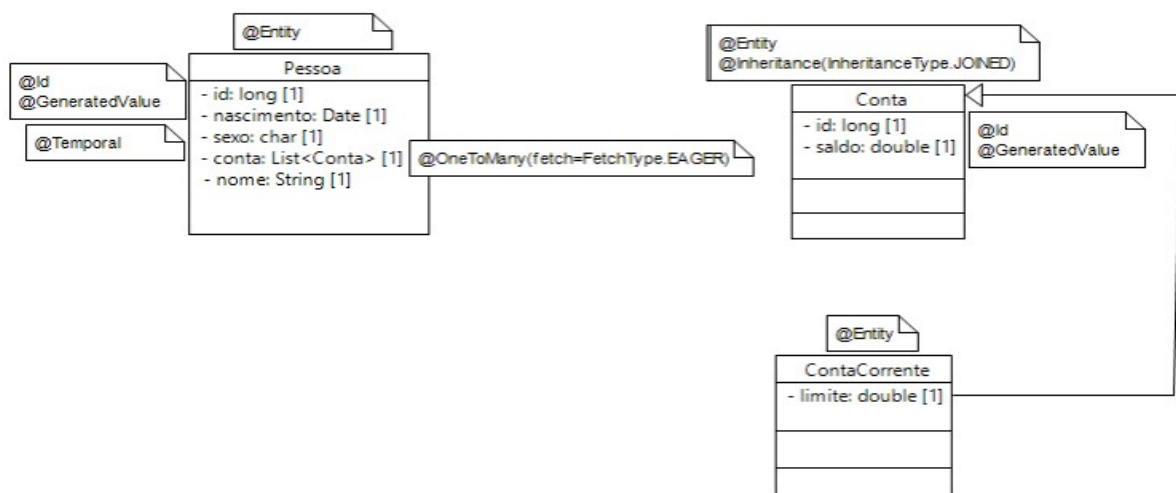


Figura 29 – Diagrama de Exemplo com Agregação.

O atributo `conta`, gerado pela associação, recebe a anotação `@OneToMany` em conjunto com a propriedade `fetch=FetchType.EAGER`. Dessa maneira, sempre que uma `Pessoa` tiver seus dados consultados no banco todas as contas a ela relacionada virão em cascata. Tal construção possui valor semântico equivalente ao de agregação da UML.

4.5 GERAÇÃO DO CÓDIGO EXECUTÁVEL

Após o fim do mapeamento-objeto relacional, a classe `ClasseControlador.java` percorre a lista de classes transformando os atributos, heranças, importes, anotações e propriedades relacionados a cada instância de classe em código Java/JPA. Para isso, utiliza as regras de mapeamento sintático que define. Um arquivo com extensão `.java` recebe os dados gerados por `ClasseControlador.java`. Para cada classe do modelo, um arquivo `.java` é gerado.

A Figura 30 mostra o código gerado pelo diagrama com associação simples, omitindo importes e métodos de acesso. Já a Figura 31 mostra o código gerado a partir do diagrama com composição. Por fim, a Figura 32 mostra o código gerado a partir do diagrama com agregação. Nos três casos, foram respeitadas as navegabilidades.

```

@Entity
public class Pessoa{

public Pessoa () {

}

@Id
@GeneratedValue
private Long id;
@Temporal(TemporalType.TIMESTAMP)
private Date nascimento;
private Character sexo;
private String nome;
@OneToMany
private List<Conta> conta;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Conta{

public Conta () {

}

@Id
@GeneratedValue
private Long id;
private Double saldo;

@Entity
public class ContaCorrente extends Conta{

public ContaCorrente () {

}

private Double limite;

```

Figura 30 – Código Gerado pelo Diagrama com Associação Simples.

```

@Entity
public class Pessoa{

public Pessoa () {

}

@Id
@GeneratedValue
private Long id;
@Temporal(TemporalType.TIMESTAMP)
private Date nascimento;
private Character sexo;
private String nome;
@OneToMany(fetch=FetchType.EAGER)
private List<Conta> conta;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Conta{

public Conta () {

}

@Id
@GeneratedValue
private Long id;
private Double saldo;

@Entity
public class ContaCorrente extends Conta{

public ContaCorrente () {

}

private Double limite;

```

Figura 31 – Código Gerado pelo Diagrama com Agregação.


```

@Entity
public class Pessoa{

public Pessoa () {
}

@Id
@GeneratedValue
private Long id;
@Temporal(TemporalType.TIMESTAMP)
private Date nascimento;
private Character sexo;
private String nome;
@OneToMany(cascade=CascadeType.ALL)
private List<Conta> conta;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Conta{

public Conta () {
}

@Id
@GeneratedValue
private Long id;
private Double saldo;

@Entity
public class ContaCorrente extends Conta{

public ContaCorrente () {
}

private Double limite;
}

```

Figura 32 – Código Gerado pelo Diagrama com Composição.

Ao final da transformação, uma mensagem é exibida na interface da aplicação informando ao usuário que o processo está finalizado (Figura 33). Têm-se fim a engenharia à frente.

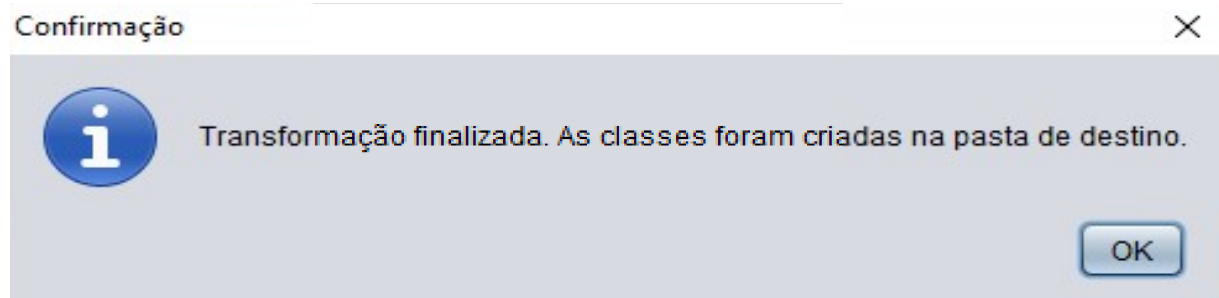


Figura 33 – *Feedback* Final.

5 CONCLUSÃO

Esta monografia apresentou a *GenJPA*, uma ferramenta de apoio à metodologias dirigidas por modelos baseada em princípios e tecnologias de *Model Driven Architecture*. Foram descritas suas funcionalidades, características e tecnologias empregadas no seu desenvolvimento.

A *GenJPA* gera código Java/JPA, automatizando a implementação das entidades da aplicação, bem como a estrutura física do banco de dados. Para isso, os conceitos da engenharia à frente foram seguidos. A transformação modelo-código na ferramenta além de mapear os blocos básicos do diagrama de classes, também abrange os conceitos mais ricos do modelo (agregação, composição e navegabilidade), diminuindo assim a ocorrência de inconsistências. A tabela 6 faz um comparativo entre a aplicação e as ferramentas citadas nos trabalhos relacionados.

Característica	Ferramentas			
	<i>Gencode</i>	<i>Hiberobjects</i>	<i>MD-JPA</i>	<i>GenJPA</i>
Independência de Ferramenta <i>CASE</i>				
Mapeamento dos Blocos Básicos (Classes, Atributos e Heranças)				
Mapeamento das Composições				
Mapeamento das Agregações				
Mantém as restrições de Navegabilidade				
Utiliza Mecanismos de Extensão da UML (Estereótipos, Valores Atribuídos)				
Funciona Apenas Atrélada a um Ambiente Específico (Eclipse)				

Tabela 6 – Comparativo entre Ferramentas

Percebe-se que a aplicação apresenta uma série de benefícios quando comparada com as demais, tais como:

- Não limita-se a um ambiente específico de execução, podendo ser usada em qualquer sistema operacional que contenha uma implementação da *Java Virtual Machine*;
- Mantém as restrições de projeto, pois consegue mapear para o código os conceitos mais ricos do diagrama de entrada, diminuindo assim as inconsistências.
- Evita que o usuário tenha que aprender novas notações da UML, já que não utiliza mecanismos de extensão para geração do código. Isso diminui o acoplamento entre modelo e código executável;
- Utiliza padrões que possibilitam ao usuário realizar a modelagem em qualquer ferramenta e utilizar seus modelos como entrada da transformação (XMI).

- Por fim, a *GenJPA* diminui o tempo de implementação de sistemas Java que façam uso de uma camada de dados. O código resultante do processo de engenharia à frente implementado na aplicação pode ser utilizado em sistemas *Web*, *Mobile* e *Desktop*.

As informações contidas nesta monografia também podem servir de ajuda para outros desenvolvedores interessados em implementar ferramentas de apoio à Métodos Dirigidos por Modelo para novos domínios de aplicação. Nos anexos, encontram-se as regras de extração dos elementos do diagrama, bem como alguns casos de estudos que apresentam melhor o funcionamento da *GenJPA*.

5.1 TRABALHOS FUTUROS

Apesar de cumprir aquilo que se propôs, algumas melhorias poderiam ser feitas na ferramenta apresentada, como:

- Mostrar o resultado das validações diretamente na interface da aplicação;
- Aumentar a quantidade de anotações da JPA suportadas, bem como as opções de mapeamento fornecidas ao usuário para o controle da transformação;
- Implementar uma versão *Web* que contemple as duas melhorias citadas.

REFERÊNCIAS

- 1 MAGALHAES, L. P. A. *Um estudo sobre a engenharia de ida e volta entre UML e java*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DE MINAS GERAIS, 2011. Citado 9 vezes nas páginas 8, 14, 15, 16, 22, 25, 26, 27 e 38.
- 2 OMG. *UML Superstructure*. 2011. <<http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>>. [Online; acessado 30-Maio-2017]. Citado 3 vezes nas páginas 8, 21 e 46.
- 3 BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML: guia do usuário*. [S.l.]: Elsevier Brasil, 2006. Citado 13 vezes nas páginas 8, 14, 15, 19, 20, 21, 22, 23, 24, 30, 31, 32 e 33.
- 4 RAMALHO, F. *PLP - Paradigma de Programação Orientado a Modelos*. 2008. UNIVERSIDADE FEDERAL DE CAMPINA GRANDE. Citado 2 vezes nas páginas 8 e 26.
- 5 ROMANATO, A. *Introdução ao Java Virtual Machine*. Citado 2 vezes nas páginas 8 e 28.
- 6 CGERNERT. *Database Development with HiberObjects*. 2008. <<https://techieexchange.wordpress.com/2008/02/07/database-development-with-hiberobjects/>>. [Online; acessado 24-Junho-2017]. Citado 2 vezes nas páginas 8 e 36.
- 7 TORRES, A. *MD-JPA: Um perfil UML para modelagem do mapeamento objeto-relacional com JPA em uma abordagem dirigida por modelos*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2009. Citado 4 vezes nas páginas 8, 29, 36 e 37.
- 8 GUEDES, G. T. *UML 2-Uma Abordagem Prática-1ª Edição*. [S.l.]: Novatec Editora, 2011. Citado 13 vezes nas páginas 14, 19, 20, 21, 22, 23, 24, 30, 31, 32, 34, 65 e 74.
- 9 GESSENHARTER, D. Mapping the uml2 semantics of associations to a java code generation model. *Model Driven Engineering Languages and Systems*, Springer, p. 813–827, 2008. Citado 2 vezes nas páginas 15 e 38.
- 10 JUNIOR, P. J. *Java: Guia do programador*. [S.l.]: Novatec Editora, 2007. Citado 2 vezes nas páginas 19 e 20.
- 11 DOBING, B.; PARSONS, J. How uml is used. *Communications of the ACM*, ACM, v. 49, n. 5, p. 109–113, 2006. Citado na página 22.
- 12 SZLENK, M. Formal semantics and reasoning about uml class diagram. In: IEEE. *Dependability of Computer Systems, 2006. DepCos-RELCOMEX'06. International Conference on*. [S.l.], 2006. p. 51–59. Citado na página 22.
- 13 PARADA, A. G.; SIEGERT, E.; BRISOLARA, L. B. de. Generating java code from uml class and sequence diagrams. In: IEEE. *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*. [S.l.], 2011. p. 99–101. Citado na página 24.
- 14 PARADA, A. G.; SIEGERT, E.; BRISOLARA, L. B. de. Gencode: A tool for generation of java code from uml class models. In: *Proc. 26th South Symposium on Microelectronics (SIM 2011)*. [S.l.: s.n.], 2011. p. 173–176. Citado 3 vezes nas páginas 24, 25 e 35.

- 15 MAIA, N. E. N. *ODYSSEY-MDA: uma abordagem para a transformação de modelos*. Dissertação (Mestrado) — UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2006. Citado na página 26.
- 16 OMG. *XML Metadata Interchange*. 2015. <<http://www.omg.org/spec/XMI/>>. [Online; acessado 01-Março-2017]. Citado na página 27.
- 17 CODINGAME. *Top Programming Languages to Learn in 2017*. 2017. <<https://www.codingame.com/blog/top-programming-languages-to-learn-in-2017/>>. [Online; acessado 15-Maio-2017]. Citado na página 27.
- 18 MENDES, D. R. *Programação Java com ênfase em Orientação a Objetos*. [S.l.]: Novatec Editora, 2009. Citado na página 27.
- 19 CLARO, D. B.; SOBRAL, J. B. M. Programação em java. *Livro programando em Java 1ª edição*, p. 12, 2008. Citado na página 27.
- 20 BRAZ, C. C. M. *Introdução à Linguagem Java*. 2005. <<http://www.ebah.com.br/content/ABAAAF70AJ/introducao-a-java>>. [Online; acessado 22-Maio-2017]. Citado na página 28.
- 21 CAELUM. *Java e Orientação a Objetos*. 2008. <<https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf>>. [Online; acessado 22-Maio-2017]. Citado 2 vezes nas páginas 28 e 29.
- 22 K19. *Persistência com JPA 2 e Hibernate*. 2015. <<http://online.k19.com.br/libraries/handouts/k21>>. [Online; acessado 30-Maio-2017]. Citado 2 vezes nas páginas 29 e 30.
- 23 ORACLE. *The Java EE 5 Tutorial*. 2010. <<http://docs.oracle.com/javase/5/tutorial/doc/bnbqa.html>>. [Online; acessado 30-Maio-2017]. Citado na página 30.
- 24 HIBEROBJECTS. *HIBEROBJECTS*. Citado na página 36.
- 25 OMG. *MDA - The Architecture Of Choice For A Changing World*. 2016. <<http://www.omg.org/mda/>>. [Online; acessado 15-Maio-2017]. Nenhuma citação no texto.
- 26 PAPYRUS. *Papyrus: Modeling Environment*. 2017. <<https://eclipse.org/papyrus/>>. [Online; acessado 30-Maio-2017]. Citado na página 40.
- 27 JDOM. *JDOM*. 2017. <<http://www.jdom.org/>>. [Online; acessado 22-Maio-2017]. Citado na página 43.

ANEXO A – REGRAS DE EXTRAÇÃO EM ACCELEO

Este anexo contém a implementação das regras de extração dos itens de interesse do diagrama de classes. O código está escrito em Acceleo, implementação da linguagem *Model To Text Language* da MDA.

```
[comment encoding = UTF-8 /]
[module main('http://www.eclipse.org/uml2/4.0.0/UML')]

[template public main(model : Model)]
    [comment @main /]
    [file (self.name+'.xml', false)]
    <elementos>
        [for(c:Class | model.packagedElement->filter(Class))]
            <classe>
                <id>[c.name/]</id>
                <nome>[c.name.normalizarClasse()/]</nome>
                [for(g:Generalization | c.generalization)]
                <heranca>
                    <general>[g.general.name/]</general>
                    <specific>[g.specific.name/]</specific>
                </heranca>
                [//for]
                [for(p:Property | c.ownedAttribute)]
                [if(p.association=null)]
                <atributo>

                <visibilidade>[p.visibility.toString()/]</visibilidade>
                <tipo>[p.type.name/]</tipo>
                <nome>[p.name.normalizarAtributo()/]</nome>
                [if(p.default<>null)]
                <default>[p.default/]</default>
                [//if]
                </atributo>
                [//if]
            [//for]
        </classe>
    [//for]

    [for(a:Association | model.packagedElement->filter(Association))]
        <associacao>
            <nome>[a.name/]</nome>
            [a.associationEnd()/]
        </associacao>
    [//for]
</elementos>
[/file]
[/template]

[query private associationEnd(a:Association):
String = a.memberEnd->iterate(p:Property; result:String='' |
if result.size() = 0 then
result.concat('<membroA>'+
    '\n\t<classe>'+p.type.name+'</classe>'+
    '\n\t<agregacao>'+p.aggregation+'</agregacao>'+
    '\n\t<minimo>'+p.lower+'</minimo>'+
    '\n\t<maximo>'+p.upper+'</maximo>'+
    if p.isNavigable() = true then
'\n\t<navegavel>true</navegavel>' else
'\n\t<navegavel>false</navegavel>' endif
    +'\n</membroA>\n'
)]
```

```
else
result.concat('<membroB>'+
    '\n\t<classe>'+p.type.name+'</classe>'+
    '\n\t<agregacao>'+p.aggregation+'</agregacao>'+
    '\n\t<minimo>'+p.lower+'</minimo>'+
    '\n\t<maximo>'+p.upper+'</maximo>'+
    if p.isNavigable() = true then
'\n\t<navegavel>'+true+'</navegavel>' else
'\n\t<navegavel>false</navegavel>' endif
    +'\n</membroB>')
endif
)
/]

[query public normalizarClasse(s:String) : String = invoke
('org.eclipse.acceleio.module.umlToJpa.validacao.NormalizadorDeNomes',
normalizarNomeClassificador(java.lang.String)',Sequence{self})/1

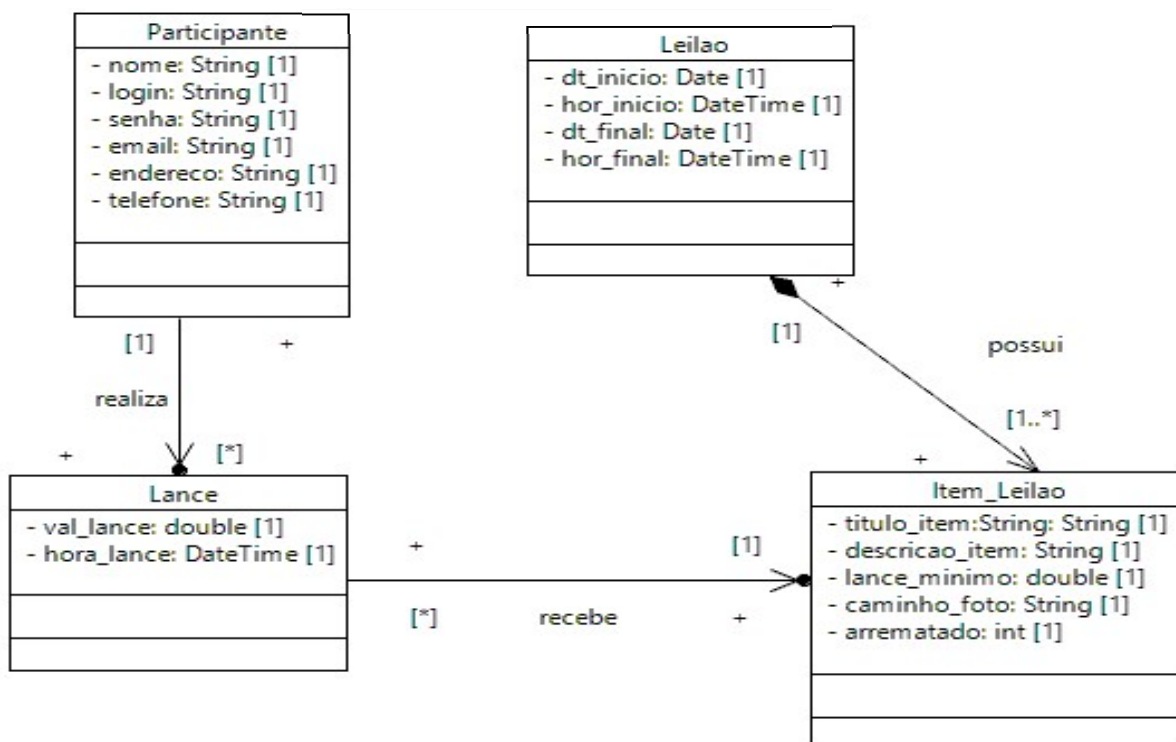
[query public normalizarAtributo(s:String) : String = invoke
('org.eclipse.acceleio.module.umlToJpa.validacao.NormalizadorDeNomes',
normalizarNomeAtributo(java.lang.String)',Sequence{self})/1

[query public normalizarMetodo(s:String) : String = invoke
('org.eclipse.acceleio.module.umlToJpa.validacao.NormalizadorDeNomes',
normalizarNomeMetodo(java.lang.String)',Sequence{self})/1
```

ANEXO B – SISTEMA DE LEILÃO ONLINE

Este anexo apresenta o código gerado pela *GenJPA* a partir de um diagrama de classes que modela um Sistema de Leilão Online [8]. As entidades geradas pela ferramenta foram utilizadas na geração de um banco de dados relacional. O diagrama Entidade-Relacionamento apresentado no final deste anexo representa o banco de dados gerado pelas classes *Beans*.

B.1 DIAGRAMA DE CLASSES



B.2 ENTIDADES

B.2.1 Participante.java

```

package modelo;

import javax.persistence.Entity;
import java.util.List;
import javax.persistence.GeneratedValue;
import javax.persistence.OneToOne;
import javax.persistence.Id;

@Entity

```



```
public class Participante{

public Participante () {
}

@Id
@GeneratedValue
private Long id;
private String nome;
private String login;
private String senha;
private String email;
private String endereco;
private String telefone;
@OneToMany
private List<Lance> lance;

public String getNome (){
    return this.nome;
}

public void setNome (String nome){
    this.nome = nome;
}

public String getLogin (){
    return this.login;
}

public void setLogin (String login){
    this.login = login;
}

public String getSenha (){
    return this.senha;
}

public void setSenha (String senha){
    this.senha = senha;
}

public String getEmail (){
    return this.email;
}
```

```
    }

    public void setEmail (String email){
        this.email = email;
    }

    public String getEndereco (){
        return this.endereco;
    }

    public void setEndereco (String endereco){
        this.endereco = endereco;
    }

    public String getTelefone (){
        return this.telefone;
    }

    public void setTelefone (String telefone){
        this.telefone = telefone;
    }

    public List<Lance> getLance (){
        return this.lance;
    }

    public void setLance (List<Lance> lance){
        this.lance = lance;
    }

    public Long getId (){
        return this.id;
    }

    public void setId (Long id){
        this.id = id;
    }

}
```

B.2.2 Lance.java

```
package modelo;

import javax.persistence.Entity;
import javax.persistence.ManyToOne;
import java.util.Calendar;
import javax.persistence.GeneratedValue;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Id;

@Entity
public class Lance{

    public Lance () {

        @Id
        @GeneratedValue
        private Long id;
        private Double val_lance;
        @Temporal(TemporalType.TIMESTAMP)
        private Calendar hora_lance;
        @ManyToOne
        private Item_Leilao item_leilao;

        public Double getVal_lance (){
            return this.val_lance;
        }

        public void setVal_lance (Double val_lance){
            this.val_lance = val_lance;
        }

        public Calendar getHora_lance (){
            return this.hora_lance;
        }

        public void setHora_lance (Calendar hora_lance){
            this.hora_lance = hora_lance;
        }

        public Item_Leilao getItem_leilao (){
            return this.item_leilao;
        }
    }
}
```

```
    }

    public void setItem_leilao (Item_Leilao item_leilao){
        this.item_leilao = item_leilao;
    }

    public Long getId (){
        return this.id;
    }

    public void setId (Long id){
        this.id = id;
    }
}
```

B.2.3 Leilao.java

```
package modelo;

import javax.persistence.Entity;
import java.util.List;
import java.util.Calendar;
import javax.persistence.CascadeType;
import javax.persistence.GeneratedValue;
import javax.persistence.Temporal;
import java.util.Date;
import javax.persistence.TemporalType;
import javax.persistence.OneToOne;
import javax.persistence.Id;

@Entity
public class Leilao{

    public Leilao () {

        @Id
        @GeneratedValue
        private Long id;
        @Temporal(TemporalType.TIMESTAMP)
        private Date dt_inicio;
        @Temporal(TemporalType.TIMESTAMP)
```

```
private Calendar hor_inicio;
@Temporal(TemporalType.TIMESTAMP)
private Date dt_final;
@Temporal(TemporalType.TIMESTAMP)
private Calendar hor_final;
@OneToMany(cascade=CascadeType.ALL)
private List<Item_Leilao> item_leilao;

public Date getDt_inicio (){
    return this.dt_inicio;
}

public void setDt_inicio (Date dt_inicio){
    this.dt_inicio = dt_inicio;
}

public Calendar getHor_inicio (){
    return this.hor_inicio;
}

public void setHor_inicio (Calendar hor_inicio){
    this.hor_inicio = hor_inicio;
}

public Date getDt_final (){
    return this.dt_final;
}

public void setDt_final (Date dt_final){
    this.dt_final = dt_final;
}

public Calendar getHor_final (){
    return this.hor_final;
}

public void setHor_final (Calendar hor_final){
    this.hor_final = hor_final;
}

public List<Item_Leilao> getItem_leilao (){
    return this.item_leilao;
}
```

```
    }

    public void setItem_leilao (List<Item_Leilao> item_leilao){
        this.item_leilao = item_leilao;
    }

    public Long getId (){
        return this.id;
    }

    public void setId (Long id){
        this.id = id;
    }
}
```

B.2.4 Item_Leilao.java

```
package modelo;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Item_Leilao{

    public Item_Leilao () {

        @Id
        @GeneratedValue
        private Long id;
        private String titulo_item;
        private String descricao_item;
        private Double lance_minimo;
        private String caminho_foto;
        private Integer arrematado;

        public String getTitulo_item (){
            return this.titulo_item;
        }
    }
}
```

```
public void setTitulo_item (String titulo_item){
    this.titulo_item = titulo_item;
}

public String getDescricao_item (){
    return this.descricao_item;
}

public void setDescricao_item (String descricao_item){
    this.descricao_item = descricao_item;
}

public Double getLance_minimo (){
    return this.lance_minimo;
}

public void setLance_minimo (Double lance_minimo){
    this.lance_minimo = lance_minimo;
}

public String getCaminho_foto (){
    return this.caminho_foto;
}

public void setCaminho_foto (String caminho_foto){
    this.caminho_foto = caminho_foto;
}

public Integer getArrematado (){
    return this.arrematado;
}

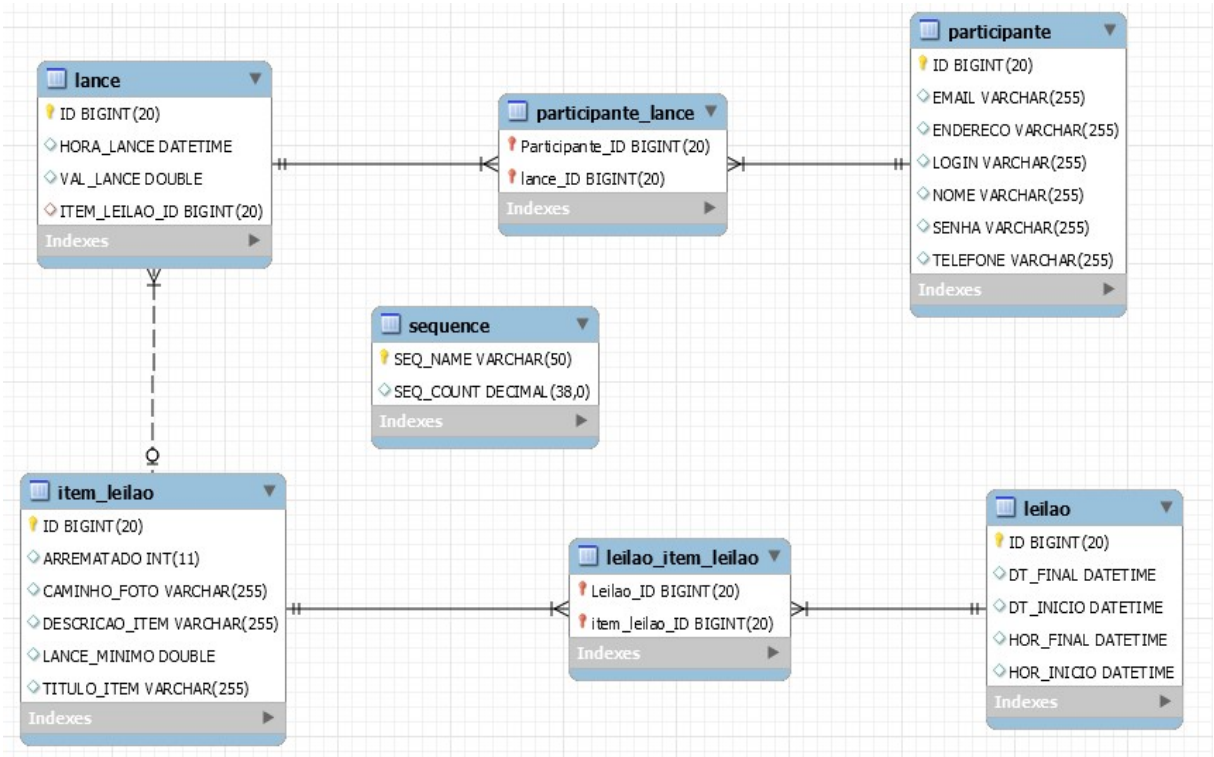
public void setArrematado (Integer arrematado){
    this.arrematado = arrematado;
}

public Long getId (){
    return this.id;
}

public void setId (Long id){
    this.id = id;
}
```

```
}  
  
}
```

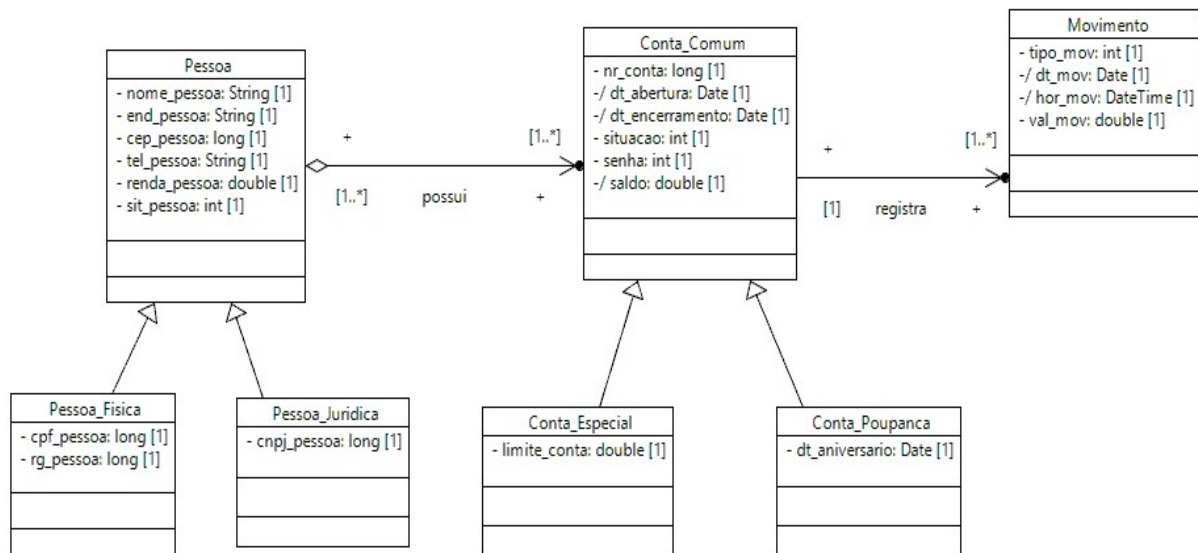
B.3 DIAGRAMA ENTIDADE-RELACIONAMENTO



ANEXO C – SISTEMA BANCÁRIO

Este anexo apresenta o código gerado pela *GenJPA* a partir de um diagrama de classes que modela um Sistema de Leilão Online [8]. As entidades geradas pela ferramenta foram utilizadas na geração de um banco de dados relacional. O diagrama Entidade-Relacionamento apresentado no final deste anexo representa o banco de dados gerado pelas classes *Beans*.

C.1 DIAGRAMA DE CLASSES



C.2 ENTIDADES

C.2.1 Conta_Comum.java

```

package modelo;

import javax.persistence.Entity;
import javax.persistence.InheritanceType;
import java.util.List;
import javax.persistence.Inheritance;
import javax.persistence.GeneratedValue;
import javax.persistence.Temporal;
import java.util.Date;
import javax.persistence.TemporalType;
import javax.persistence.OneToOne;
import javax.persistence.Id;
  
```

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Conta_Comum{

    public Conta_Comum () {
    }

    @Id
    @GeneratedValue
    private Long id;
    private Long nr_conta;
    @Temporal(TemporalType.TIMESTAMP)
    private Date dt_abertura;
    @Temporal(TemporalType.TIMESTAMP)
    private Date dt_encerramento;
    private Integer situacao;
    private Integer senha;
    private Double saldo;
    @OneToMany
    private List<Movimento> movimento;

    public Long getNr_conta () {
        return this.nr_conta;
    }

    public void setNr_conta (Long nr_conta) {
        this.nr_conta = nr_conta;
    }

    public Date getDt_abertura () {
        return this.dt_abertura;
    }

    public void setDt_abertura (Date dt_abertura) {
        this.dt_abertura = dt_abertura;
    }

    public Date getDt_encerramento () {
        return this.dt_encerramento;
    }

    public void setDt_encerramento (Date dt_encerramento) {
        this.dt_encerramento = dt_encerramento;
    }
}
```

```
}

public Integer getSituacao (){
    return this.situacao;
}

public void setSituacao (Integer situacao){
    this.situacao = situacao;
}

public Integer getSenha (){
    return this.senha;
}

public void setSenha (Integer senha){
    this.senha = senha;
}

public Double getSaldo (){
    return this.saldo;
}

public void setSaldo (Double saldo){
    this.saldo = saldo;
}

public List<Movimento> getMovimento (){
    return this.movimento;
}

public void setMovimento (List<Movimento> movimento){
    this.movimento = movimento;
}

public Long getId (){
    return this.id;
}

public void setId (Long id){
    this.id = id;
}
```

```
}
```

C.2.2 Conta_Poupanca.java

```
package modelo;

import javax.persistence.Entity;
import javax.persistence.ManyToOne;
import java.util.Calendar;
import javax.persistence.GeneratedValue;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Id;

@Entity
public class Lance{

    public Lance () {

    }

    @Id
    @GeneratedValue
    private Long id;
    private Double val_lance;
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar hora_lance;
    @ManyToOne
    private Item_Leilao item_leilao;

    public Double getVal_lance (){
        return this.val_lance;
    }

    public void setVal_lance (Double val_lance){
        this.val_lance = val_lance;
    }

    public Calendar getHora_lance (){
        return this.hora_lance;
    }

    public void setHora_lance (Calendar hora_lance){
        this.hora_lance = hora_lance;
    }
}
```

```
    }

    public Item_Leilao getItem_leilao () {
        return this.item_leilao;
    }

    public void setItem_leilao (Item_Leilao item_leilao) {
        this.item_leilao = item_leilao;
    }

    public Long getId () {
        return this.id;
    }

    public void setId (Long id) {
        this.id = id;
    }
}
```

C.2.3 Conta_Especial.java

```
package modelo;

import javax.persistence.Entity;

@Entity
public class Conta_Especial extends Conta_Comum {

    public Conta_Especial () {
    }

    private Double limite_conta;

    public Double getLimite_conta () {
        return this.limite_conta;
    }

    public void setLimite_conta (Double limite_conta) {
        this.limite_conta = limite_conta;
    }
}
```

C.2.4 Pessoa.java

```
package modelo;

import javax.persistence.Entity;
import javax.persistence.InheritanceType;
import java.util.List;
import javax.persistence.Inheritance;
import javax.persistence.GeneratedValue;
import javax.persistence.FetchType;
import javax.persistence.ManyToMany;
import javax.persistence.Id;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa{

    public Pessoa () {

        @Id
        @GeneratedValue
        private Long id;
        private String nome_pessoa;
        private String end_pessoa;
        private Long cep_pessoa;
        private String tel_pessoa;
        private Double renda_pessoa;
        private Integer sit_pessoa;
        @ManyToMany(fetch=FetchType.EAGER)
        private List<Conta_Comum> conta_comum;

        public String getNome_pessoa () {
            return this.nome_pessoa;
        }

        public void setNome_pessoa (String nome_pessoa) {
            this.nome_pessoa = nome_pessoa;
        }

        public String getEnd_pessoa () {
            return this.end_pessoa;
        }
    }
}
```

```
public void setEnd_pessoa (String end_pessoa){
    this.end_pessoa = end_pessoa;
}

public Long getCep_pessoa (){
    return this.cep_pessoa;
}

public void setCep_pessoa (Long cep_pessoa){
    this.cep_pessoa = cep_pessoa;
}

public String getTel_pessoa (){
    return this.tel_pessoa;
}

public void setTel_pessoa (String tel_pessoa){
    this.tel_pessoa = tel_pessoa;
}

public Double getRenda_pessoa (){
    return this.renda_pessoa;
}

public void setRenda_pessoa (Double renda_pessoa){
    this.renda_pessoa = renda_pessoa;
}

public Integer getSit_pessoa (){
    return this.sit_pessoa;
}

public void setSit_pessoa (Integer sit_pessoa){
    this.sit_pessoa = sit_pessoa;
}

public List<Conta_Comum> getConta_comum (){
    return this.conta_comum;
}

public void setConta_comum (List<Conta_Comum> conta_comum){
```

```
        this.conta_comum = conta_comum;
    }

    public Long getId () {
        return this.id;
    }

    public void setId (Long id) {
        this.id = id;
    }
}
```

C.2.5 Pessoa_Juridica.java

```
package modelo;

import javax.persistence.Entity;

@Entity
public class Pessoa_Juridica extends Pessoa {

    public Pessoa_Juridica () {
    }

    private Long cnpj_pessoa;

    public Long getCnpj_pessoa () {
        return this.cnpj_pessoa;
    }

    public void setCnpj_pessoa (Long cnpj_pessoa) {
        this.cnpj_pessoa = cnpj_pessoa;
    }
}
```

C.2.6 Pessoa_Fisica.java

```
package modelo;

import javax.persistence.Entity;

@Entity
```



```
public class Pessoa_Fisica extends Pessoa{

public Pessoa_Fisica () {
}

    private Long cpf_pessoa;
    private Long rg_pessoa;

    public Long getCpf_pessoa (){
        return this.cpf_pessoa;
    }

    public void setCpf_pessoa (Long cpf_pessoa){
        this.cpf_pessoa = cpf_pessoa;
    }

    public Long getRg_pessoa (){
        return this.rg_pessoa;
    }

    public void setRg_pessoa (Long rg_pessoa){
        this.rg_pessoa = rg_pessoa;
    }

}
```

C.2.7 Movimento.java

```
package modelo;

import javax.persistence.Entity;
import java.util.Calendar;
import javax.persistence.GeneratedValue;
import javax.persistence.Temporal;
import java.util.Date;
import javax.persistence.TemporalType;
import javax.persistence.Id;

@Entity
public class Movimento{

public Movimento () {
}

}
```

```
@Id
@GeneratedValue
private Long id;
private Integer tipo_mov;
@Temporal(TemporalType.TIMESTAMP)
private Date dt_mov;
@Temporal(TemporalType.TIMESTAMP)
private Calendar hor_mov;
private Double val_mov;

public Integer getTipo_mov () {
    return this.tipo_mov;
}

public void setTipo_mov (Integer tipo_mov) {
    this.tipo_mov = tipo_mov;
}

public Date getDt_mov () {
    return this.dt_mov;
}

public void setDt_mov (Date dt_mov) {
    this.dt_mov = dt_mov;
}

public Calendar getHor_mov () {
    return this.hor_mov;
}

public void setHor_mov (Calendar hor_mov) {
    this.hor_mov = hor_mov;
}

public Double getVal_mov () {
    return this.val_mov;
}

public void setVal_mov (Double val_mov) {
    this.val_mov = val_mov;
}
```

```

public Long getId (){
    return this.id;
}

public void setId (Long id){
    this.id = id;
}

}

```

C.3 DIAGRAMA ENTIDADE-RELACIONAMENTO

