UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

FERRAMENTA PARA GERAÇÃO DE CÓDIGO A PARTIR DA ESPECIALIZAÇÃO DO DIAGRAMA DE CLASSES

ALEXANDRO DESCHAMPS

BLUMENAU
2005

2005/1-01

ALEXANDRO DESCHAMPS

FERRAMENTA PARA GERAÇÃO DE CÓDIGO A PARTIR DA ESPECIALIZAÇÃO DO DIAGRAMA DE CLASSES

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Sistemas de Informação – Bacharelado.

Prof. Everaldo Artur Grahl – Orientador

BLUMENAU
2005

2005/1-01

FERRAMENTA PARA GERAÇÃO DE CÓDIGO A PARTIR DA ESPECIALIZAÇÃO DO DIAGRAMA DE CLASSES

Por

ALEXANDRO DESCHAMPS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Everaldo Artur Grahl – Orientador, FURB
Membro:	Prof. Marcel Hugo, FURB
Membro:	Prof. Alexander Roberto Valdameri, FURB

Blumenau, 06 de Julho de 2005.

Dedico este trabalho a todas as pessoas que fazem parte da minha vida, mesmo que em pensamento. E a todos que de alguma forma contribuíram para a realização do mesmo.

Se você deseja conhecer uma pessoa de verdade, de o poder a ela.

Autor Desconhecido

AGRADECIMENTOS

À Deus, por tudo e por todos. Sem fé, nada é possível!

Ao meu pai Carlos, pela motivação e persistência.

À minha mãe Luzia, por todo apoio, vitórias e por seus valiosos conselhos.

À minha irmã Adriana, pela parceria em todos os momentos.

À minha namorada Leila por seu amor, dedicação e incondicional apoio.

A toda minha família, por todo amor, carinho e compreensão.

Aos professores, por todo o conhecimento passado durante o curso.

Ao meu orientador Everaldo, por acreditar na conclusão do presente trabalho.

Aos “manos” Dalmarco e Tambosi, pela amizade e apoio durante o curso.

A todos os meus colegas de curso, por tudo!

Aos meus amigos, pessoas com que posso contar nos bons e maus momentos.

A Elinton e Everton Marçal, por serem os primeiros a acreditarem no meu potencial.

À Ricardo de Freitas Becker, pela oportunidade e por todo conhecimento passado.

À Santa Catarina Informática, Tron Informática e a Mult Sistemas.

Ao Instituto GENE e todos seus colaboradores por acreditarem neste trabalho.

Às dificuldades, por proporcionarem mudanças significativas em minha vida.

RESUMO

Uma das grandes preocupações das organizações desenvolvedoras de softwares é a obtenção de certificações que atestem a qualidade dos seus produtos. Uma das ênfases abordadas pelos Sistemas de Garantia da Qualidade é a definição do processo de desenvolvimento de software das organizações. Para obter um processo de desenvolvimento de software bem definido, as organizações utilizam as metodologias abordadas pela Engenharia de Software em conjunto com o uso de ferramentas que automatizem este processo. Este trabalho tem a finalidade de apresentar a construção de uma ferramenta de auxílio ao processo de desenvolvimento de software que possibilita a especialização dos diagramas de classes UML (*Unified Modeling Language*) por arquitetura e idioma. A finalidade desta especialização é a geração de código através do uso de *plugins*. A ferramenta proposta ajuda a agregar uma melhor padronização ao processo de desenvolvimento, diminui o tempo de implementação e aumenta a qualidade dos softwares desenvolvidos.

Palavras chaves: UML; Diagrama de Classes; Geração de Código, Internacionalização.

ABSTRACT

One of the great concerns of organizations that develop software is the attainment of certifications that guarantee the quality of its products. One of the boarded emphases for System of Quality Guaranty is the definition of the process of development of software of the organizations. To get a process of development of software well defined, the organizations use the boarded methodologies for the Engineering of Software in set with the use of tools that automate this process. This work has the purpose to present the construction of a tool of aid to the process of software development that makes possible the specialization of the diagrams of class UML (Unified Modeling Language) for architecture and language. The purpose of this specialization is the generation of code through the use of plugins. The tool proposal helps to add a better standardization to the development process, decrease the implementation time and increase the developed quality of software.

Keys-words: UML; Class Diagram; Code Generation, Internationalization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo da notação de um pacote....20
Figura 2 – Exemplo da notação de uma classe....21
Quadro 1 – Especificação de atributo....23
Quadro 2 – Especificação de operação....24
Quadro 3 – Especificação de parâmetro....26
Figura 3 – Pacotes dos casos de uso....35
Figura 4 – Casos de uso das estruturas básicas....36
Figura 5 – Casos de uso das estruturas de especialização....37
Figura 6 – Casos de uso do modelo de classes....38
Figura 7 – Diagrama de classes de domínio....39
Figura 8 – Pacotes do diagrama de classes de especificação....40
Figura 9 – Diagrama de classes da camada de persistência....40
Figura 10 – Diagrama de classes da camada de lógica....41
Figura 11 – Diagrama de classes da camada de apresentação....42
Figura 12 – Diagrama de atividades....43
Figura 13 – Tipos de dado das propriedades de especialização....44
Figura 14 – Exemplo de registros de propriedades de especialização....49
Quadro 4 – Trecho do conteúdo de um artefato XMI....50
Quadro 5 – Interface para construção de um *plugin*....51
Quadro 6 – Exemplo de uma classe básica para construção de um *plugin*....52
Figura 15 – Estrutura da pasta de implantação da ferramenta....52
Figura 16 – *Menu* de manutenções....53
Figura 17 – *Menu* do modelo de classes....54
Figura 18 – Manutenção de projetos....54
Figura 19 – Manutenção de arquiteturas....55
Figura 20 – Manutenção de idiomas....55
Figura 21 – Manutenção de tipos de dado....56
Figura 22 – Manutenção de grupos de propriedades....56
Figura 23 – Manutenção da especialização de pacotes....57
Figura 24 – Manutenção da especialização de interfaces....58
Figura 25 – Manutenção da especialização de classes....59
Figura 26 – Manutenção da especialização de atributos....60
Figura 27 – Manutenção da especialização de operações....61
Figura 28 – Manutenção da especialização de parâmetros....61
Figura 29 – Manutenção da especialização de relacionamentos....62
Figura 30 – Especialização dos modelos de classes....63
Figura 31 – Geração de código....63
Figura 32 – Diagrama de localizações....64
Quadro 7 – Trecho do código escrito para o *plugin* de geração da arquitetura Java....65
Figura 33 – Exemplo da especialização do tipo de dado “Inteiro”....66
Quadro 8 – Trecho de código Java gerado para a classe “Estado”....66
Quadro 9 – Trecho do código escrito para o *plugin* de geração da arquitetura PCL....67
Quadro 10 – Trecho do código PCL gerado para a subclasse “Atributos” da classe “Estado” 68
Quadro 11 – Trecho do código PCL gerado para a subclasse “Visoes” da classe “Estado”....68
Quadro 12 – Código PCL gerado para a subclasse “Relacionamentos” da classe “Estado”....69
Figura 34 – Propriedades de especialização das classes de entidade....70
Figura 35 – Especialização da classe “Estado”....70
Figura 36 – Especialização da propriedade “Descrição” para o idioma inglês....71

Quadro 13 – Trecho do código PCL gerado para a classe “Estado” 72
Figura 37 – Exemplo de interface PCL gerada para manutenção da classe “Estado”..............72

LISTA DE TABELAS

Tabela 1 – Simbologia para representar multiplicidades..28
Tabela 2 – Conectividades X Multiplicidades...28
Tabela 3 – Requisitos funcionais...34
Tabela 4 – Requisitos não funcionais...34
Tabela 5 – Tipos de visibilidade das propriedades de especialização.....................................45
Tabela 6 – Possíveis descendências por tipo de visibilidade...46
Tabela 7 – Ferramentas utilizadas na especificação e implementação....................................48
Tabela 8 – Tecnologias utilizadas na especificação e implementação....................................49
Tabela 9 – Propriedades compostas...77

LISTA DE SIGLAS

ASP – *Active Server Pages*

BNF – *Backus-Naur Form*

CASE – *Computer Aided Software Engineering*

CDL – *Class Definition Language*

CMM – *Capability Maturity Model*

DLL – *Dynamic Link Library*

DTD – *Document Type Definition*

EXE – *Executable*

IE – *Information Engineering*

ISO – *International Organization for Standardization*

OMG – *Object Management Group*

OMT – *Object Modeling Technique*

OO – *Object Oriented*

PCL – *PHP Custom Library*

PHP – *Hypertext Preprocessor*

SGBD – Sistema Gerenciador de Banco de Dados

SGML – *Standart Generalized Markup Language*

UML – *Unified Modeling Language*

W3C – *World Wide Web Consortium*

XMI – *XML Metadata Interchange*

XML – *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO **14**
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 16
2 FUNDAMENTAÇÃO TEÓRICA **17**
2.1 ARQUITETURA 17
2.2 UML (*UNIFIED MODELING LANGUAGE*) 17
2.2.1 MODELO DE CLASSES 18
2.2.2 DIAGRAMA DE CLASSES 19
2.2.2.1 Pacotes 20
2.2.2.2 Interfaces 20
2.2.2.3 Classes 21
2.2.2.4 Atributos 22
2.2.2.5 Operações 24
2.2.2.6 Parâmetros 26
2.2.2.7 Relacionamentos 26
2.3 XML (*EXTENSIBLE MARKUP LANGUAGE*) 29
2.4 FERRAMENTAS CASE (*COMPUTER AIDED SOFTWARE ENGINEERING*) 30
2.4.1 XMI (*XML METADATA INTERCHANGE*) 30
2.4.2 GERADORES DE CÓDIGO 31
2.5 INTERNACIONALIZAÇÃO 31
2.6 TRABALHOS CORRELATOS 32
3 DESENVOLVIMENTO DO TRABALHO **34**
3.1 REQUISITOS DA FERRAMENTA 34
3.2 ESPECIFICAÇÃO 34
3.2.1 DIAGRAMA DE CASOS DE USO 35
3.2.2 DIAGRAMA DE CLASSES 38
3.2.3 DIAGRAMA DE ATIVIDADES 42
3.3 IMPLEMENTAÇÃO 43
3.3.1 ESTRUTURA DE ESPECIALIZAÇÃO 43
3.3.1.1 Categorias de especialização e suas propriedades 44
3.3.1.2 Utilização das categorias de especializações 46
3.3.2 TÉCNICAS E FERRAMENTAS UTILIZADAS 46

3.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO 53
3.3.3.1 Geração de código e *plugins* 64
3.4 RESULTADOS E DISCUSSÃO 73
4 CONCLUSÕES 75
4.1 EXTENSÕES 76
REFERÊNCIAS BIBLIOGRÁFICAS 78
APÊNDICE A – Descrição dos casos de uso 80
ANEXO A – Exemplos de código gerado para arquitetura Java 101
ANEXO B – Exemplo de código PCL (*PHP Custom Library*) gerado 103
ANEXO C – Exemplos de interfaces PCL (*PHP Custom Library*) geradas 107

1 INTRODUÇÃO

Uma das grandes cobranças do mercado em relação às organizações desenvolvedoras de software é a obtenção de certificações que atestem a qualidade dos softwares desenvolvidos. Uma das principais ênfases abordadas pelos Sistemas de Garantia da Qualidade é a definição do processo de desenvolvimento de software da organização. Um dos aspectos abordados pelo processo de desenvolvimento de software e de suma importância para o mesmo é a codificação. Uma codificação construída sem o uso de metodologias e padrões se torna um agravante para os quesitos tempo e qualidade, dificultando também a manutenção do software em questão.

Os grandes avanços tecnológicos são facilmente constatados através do lançamento de novas tecnologias como arquiteturas e ambientes de desenvolvimento. Estes avanços tornam a codificação dos softwares cada vez mais voláteis. Na grande maioria dos casos os avanços tecnológicos forçam as organizações desenvolvedoras de software a reescreverem seu código (SCHMIDT, 2001, p. 1). Codificar um software requer um grande conhecimento da arquitetura utilizada. Mesmo tendo em mãos um ótimo modelo, o trabalho de codificação ainda é algo extremamente demorado e na maioria das vezes repetitivo (PAULA FILHO, 2003, p. 180).

A UML (*Unified Modeling Language*) é atualmente a linguagem para documentação e modelagem de software mais difundida entre os desenvolvedores. Através dos modelos e diagramas propostos pela UML pode-se obter diferentes visões de um software, dentre eles, um diagrama que recebe um destaque especial é o diagrama de classes. O diagrama de classes é importante não só para visualização, especificação e para a documentação de modelos estruturais, mas também para a construção de sistemas executáveis por intermédio da Engenharia da Produção (*Forward Engineering*) (BOOCH, 2000, p. 104). A Engenharia da Produção é o processo de transformar um modelo em código (BOOCH, 2000, p. 112). Muitas das ferramentas CASE (*Computer Aided Software Engineering*) existentes no mercado contemplam o processo de Engenharia da Produção, mas geralmente seus recursos são extremamente limitados e não permitem ao desenvolvedor realizar a geração de código de uma forma mais refinada, não atendendo desta forma também as particularidades dos diferentes processos de desenvolvimento de software.

Um exemplo de refinamento que se torna evidente quando se leva em consideração a internacionalização de software é a necessidade da criação de softwares que suportem diferentes idiomas. Utilizando somente a especificação do diagrama de classes proposto pela UML isto não é possível. Outro refinamento que se pode citar é a utilização de um mesmo modelo de classes para geração de código em diferentes arquiteturas.

Dentro deste contexto o objetivo deste trabalho é apresentar o desenvolvimento de uma ferramenta de auxílio ao processo de desenvolvimento que permita a especialização das informações contidas em diagramas de classes UML por arquitetura e idioma, visando enriquecer desta maneira o processo de Engenharia da Produção.

A geração de código proposta pelo trabalho ocorre através da utilização de *plugins*. Os *plugins* de geração são módulos DLL (*Dynamic Link Library*) encarregados de montar a sintaxe do código, estando os mesmos ligados a uma determinada arquitetura. As informações contidas nos diagramas de classes e suas especializações são disponibilizadas através de uma interface padrão utilizada para captação de dados pelos *plugins*.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver a ferramenta de auxílio ao processo de desenvolvimento que permita aos desenvolvedores de software o enriquecimento das informações contidas em diagramas de classes UML (*Unified Modeling Language*) para geração de código. O enriquecimento proposto ocorre através da especialização dos elementos (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros e Relacionamentos) dos diagramas de classes por arquitetura e idioma.

Os objetivos específicos do trabalho são:

a) acessar e interpretar diagramas de classes através de artefatos XMI (*XML Metadata Interchange)*;

b) possibilitar a especialização dos elementos dos diagramas de classes importados por arquitetura e idioma, de forma que enriqueçam a geração de código;

c) disponibilizar uma interface de saída para *plugins* que contenham a lógica de geração de código;

d) criar rotinas que facilitem o acesso às informações dos diagramas de classes e suas respectivas especializações, visando tornar a lógica de geração a mais completa possível.

1.2 ESTRUTURA DO TRABALHO

A estrutura do trabalho está dividida em quatro capítulos: o primeiro capítulo refere-se à introdução, os objetivos e a estrutura do trabalho.

O segundo capítulo refere-se à fundamentação teórica, onde são abordados temas relevantes ao presente trabalho, juntamente com a citação de trabalhos correlatos.

O terceiro capítulo refere-se ao desenvolvimento do trabalho. Para um melhor entendimento este capítulo foi dividido em requisitos do sistema, especificação, implementação, técnicas e ferramentas utilizadas, operacionalidade da implementação e resultados.

O quarto e último capítulo refere-se às conclusões finais e extensões, incluindo também sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda alguns assuntos relevantes ao tema do presente trabalho, apresentando considerações sobre arquiteturas, UML, mais especificamente o modelo e o diagrama de classes, XML, ferramentas CASE, XMI e geradores de código.

2.1 ARQUITETURA

A arquitetura associa as capacidades do software identificadas na especificação de requisitos com os componentes do sistema que irão implementá-las. Os componentes são geralmente módulos e a arquitetura também descreve as interconexões entre eles. Além disso, a arquitetura define operadores que criam sistemas a partir de subsistemas (PFLEEGER, 2004, p. 164). Quando se fala de arquitetura de software, está se referindo as diferentes visões que possibilitam decisões sobre a construção de um software. Essas visões são formadas por um conjunto de modelos de softwares, dentre eles pode-se citar o modelo de caso de uso, análise, projeto, implementação, implantação e teste (MEDEIROS, 2004, p. 49).

A arquitetura de um software pode ser estudada de acordo com diversas perspectivas. Uma dessas perspectivas permite visualizar o sistema de software como um conjunto de camadas, onde uma camada é uma coleção de unidades de software que podem ser executadas ou acessadas. A divisão de um software em camadas permite que este seja mais portável e modificável. Uma mudança em uma camada mais baixa que não afete a sua interface não implicará em mudanças nas camadas mais altas (BEZERRA, 2003, p. 246). Na definição da arquitetura, deve-se buscar um equilíbrio entre o atendimento de requisitos atuais e futuros. A definição de estruturas e mecanismos genéricos demais produz softwares ineficientes e fora do prazo, enquanto a definição de estruturas e mecanismos específicos demais produz softwares de manutenção e extensão difíceis (PAULA FILHO, 2003, p. 126).

Neste trabalho a arquitetura foi estudada para definição da estrutura de especialização. Este estudo tem o objetivo de conhecer as características, semelhanças e diferenças das arquiteturas utilizadas para a implementação de um determinado software.

2.2 UML (*UNIFIED MODELING LANGUAGE*)

Booch (2000, p. 13) define a UML como uma linguagem destinada a visualizar, especificar, construir e documentar os artefatos de um sistema complexo de software. Outra

definição que complementa a anteriormente citada é a de Rezende (1999, p. 207), que define a UML como uma linguagem de modelagem que procura através de diagramas definir as principais fases dentro das diversas metodologias baseadas em objetos, tais como as fases de análise, projeto e implementação. O mesmo pode-se constatar em Furlan (1998, p. 38), que destaca a UML como uma linguagem de modelagem, não uma metodologia.

Atualmente a UML está em sua versão 2.0 e é composta por treze diagramas: diagrama de atividades, diagrama de caso de uso, diagrama de classe, diagrama de objetos, diagrama de seqüência, diagrama de comunicação, diagrama de estado, diagrama de pacotes, diagrama de componentes, diagrama de implantação, diagrama de interação, diagrama de tempo e o diagrama de estrutura composta (GUEDES, 2004, p. 256).

A UML foi aprovada como padrão pela OMG (*Object Management Group*) em 1997 e a sua construção teve muitos contribuintes, mas os principais atores no processo foram Grady Booch, James Rumbaugh e Ivar Jacobson. Esses três pesquisadores são freqüentemente chamados de "os três amigos". No processo de definição da UML, procurou-se aproveitar o melhor das características das notações preexistentes. A UML é independente tanto de linguagens de programação quanto de processos de desenvolvimento. Isso quer dizer que a UML pode ser utilizada para modelagem de softwares, não importa qual linguagem de programação será utilizada na implementação do software, ou qual a forma (processo) de desenvolvimento adotado. Esse é um fator importante para a utilização da UML, pois diferentes sistemas de software requerem diferentes abordagens de desenvolvimento (BEZERRA, 2003, p. 13). O diagrama de classe é considerado um dos principais diagramas da UML, pode-se ter uma visão mais completa deste diagrama nos itens a seguir.

2.2.1 MODELO DE CLASSES

Conforme estudos apresentados por Bezerra (2003, p. 96) o modelo de classes é composto pelo diagrama de classes e pela descrição textual associada. Nota-se que o modelo de classes evolui durante as iterações do desenvolvimento do software. A medida que o software é desenvolvido, o modelo de classes é incrementado com novos detalhes. Há três níveis sucessivos de abstração pelos qual o modelo de classes passa. Esses níveis são: domínio, especificação e implementação.

O modelo de classes de domínio representa as classes no domínio de negócio em questão, sendo construído na fase de análise. Por definição, um modelo de classes de domínio não leva em consideração restrições inerentes à tecnologia a ser utilizada na solução de um problema.

O modelo de classes de especificação é uma extensão do modelo de classes de domínio. Essa extensão é feita através da adição de detalhes específicos conforme a solução de software escolhida. Além disso, nesse nível são definidas novas classes necessárias para desenvolver a solução do problema. O modelo de especificação é construído na atividade de projeto de desenvolvimento de uma iteração de desenvolvimento.

O modelo de classes de implementação é uma extensão do modelo de especificação. Esse modelo corresponde à implementação das classes em alguma linguagem de programação, normalmente uma linguagem orientada a objetos. O modelo de implementação é construído na atividade de implementação de uma iteração do desenvolvimento.

2.2.2 DIAGRAMA DE CLASSES

O diagrama de classes é utilizado na construção do modelo de classes desde o nível de análise até o nível de especificação. De todos os diagramas da UML, esse é o mais rico em termos de notação (BEZERRA, 2003, p. 97). Os diagramas de classes são importantes não só para visualização, a especificação e a documentação de modelos estruturais, mas também para a construção de sistemas executáveis por intermédio da Engenharia da Produção (*Forward Engineering*) (BOOCH, 2000, p. 104). Para Furlan (1998, p. 91) o diagrama de classes é a essência da UML, resultado de uma combinação de diagramas propostos pela OMT (*Object Modeling Technique*), Booch e vários outros métodos. Um diagrama de classes mostra um conjunto de classes, interfaces, colaborações e seus relacionamentos (BOOCH, 2000, p. 104).

O estudo do diagrama de classes e de seus elementos é fundamental para definição da estrutura de especialização proposta pelo trabalho. Através deste estudo é possível determinar a interação de cada elemento do diagrama de classes, seus atributos e comportamentos. Pode-se ter uma visão mais completa dos mesmos nos itens a seguir.

2.2.2.1 Pacotes

A possibilidade de juntar classes que tenham a mesma finalidade é conhecida como empacotamento. A criação de pacotes permite uma maior organização entre as classes que são utilizadas no transcorrer da construção de um software (MEDEIROS, 2004, p. 105).

Os pacotes são utilizados para organizar seus elementos de modelagem em conjuntos maiores que possam ser manipulados como grupos (BOOCH, 2000, p. 167). É possível controlar a visibilidade desses elementos, permitindo que alguns destes itens fiquem visíveis fora do pacote, enquanto os outros ficarão ocultos. Os pacotes também podem ser empregados para apresentar diferentes visões da arquitetura de um determinado software. A notação de um pacote é a de uma pasta com uma aba (BEZERRA, 2003, p. 69). Um exemplo da notação de um pacote pode ser vista na Figura 1 .

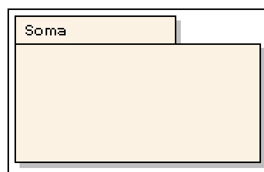


Figura 1 – Exemplo da notação de um pacote

2.2.2.2 Interfaces

Uma interface corresponde a um conjunto de especificações de serviços. Uma interface guarda algumas semelhanças com o conceito de classe abstrata. Assim como esta última, uma interface possui um nome. Da mesma forma que uma classe abstrata, uma interface não gera instâncias. Mas, ao contrário de uma classe abstrata, uma interface não contém estrutura interna (não tem atributos e nem associações). Além disso, todas as operações de uma interface só possuem especificações; a implementação (método) de cada operação não é definida na interface (BEZERRA, 2003, p. 237).

Quando um classificador implementa uma ou mais operações especificadas em uma interface, diz-se que o classificador realiza a interface. Um classificador pode realizar uma ou mais interfaces. Uma interface pode ser realizada por um ou mais classificadores (BEZERRA, 2003, p. 237).

2.2.2.3 Classes

As classes são os blocos de construção mais importantes de qualquer software orientado a objetos. Uma classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica (BOOCH, 2000, p. 47).

Medeiros (2004, p. 75) diferencia as classes de objetos utilizando como exemplo a construção de um prédio de apartamentos: para construção os apartamentos são rigorosamente iguais, baseados em uma planta que anteriormente já esclarecia o número de dormitórios, portas, louças sanitárias, fiação elétrica e disposição hidráulica. Sendo assim, a planta dos apartamentos é a classe e os apartamentos são os objetos baseados nela.

A notação de uma classe é representada por três compartimentos em um quadrado. O primeiro compartimento é reservado ao nome da classe, o segundo, a suas características ou atributos. O último compartimento é reservado a suas operações. Esta notação não é uma invenção da UML; há muitos anos se usa este formato para representar uma classe. Todas as ferramentas que permitem a construção de um diagrama de classes utilizam esta notação (GUEDES, 2004, p. 38).

Um exemplo da notação de uma classe pode ser vista na Figura 2 .

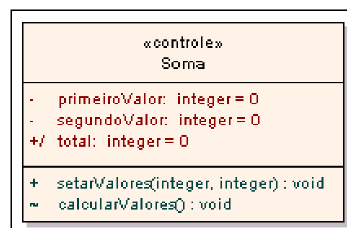


Figura 2 – Exemplo da notação de uma classe

A visibilidade de uma classe pode ser marcada por quatro tipos: *public*, *private*, *protected* ou *package*. A visibilidade *public* permite que qualquer outra classe acesse seus atributos e operações. Quando marcada com o tipo de visibilidade *private* a classe é declarada como parte de uma outra classe e somente a mesma terá acesso a seus atributos e operações. Quando marcada com o tipo de visibilidade *protected* somente poderão acessar os atributos e operações da classe as suas subclasses. Quando marcada com a visibilidade *package* os atributos e operações da classe poderão ser acessados somente por classes que pertencem ao mesmo pacote (OBJECT MANAGEMENT GROUP, 2004).

Os modificadores de uma classe são classificados como: *abstract*, *final*, *root* e *active*. O modificador *abstract* indica que a classe não poderá corresponder a objetos instanciados, servindo somente para modelo e apoio a suas classes descendentes. O modificador *final* indica que a classe não poderá ter subclasses, portanto uma classe não poderá conter os modificadores *abstract* e *final* juntos. O modificador *root* indica que a classe não possui ancestral. O modificador *active* indica que uma classe poderá ter objetos com a sua própria *thread* ou segmento de controle (MEDEIROS, 2004, p. 86).

Costuma-se categorizar as classes de um software de acordo com o tipo de responsabilidade a ela atribuída. Esta categorização foi proposta por Ivar Jacobson em uma técnica denominada Análise de Robustez. Segundo ele, as classes das quais um software é composto podem ser divididas de acordo com suas responsabilidades em classes de entidade, classes de controle e classes de fronteira (BEZERRA, 2003, p. 111).

A seguir é apresentada a definição das categorias de classes conforme estudos apresentados por Paula Filho (2003, p. 99).

Classes de entidade modelam a informação persistente, sendo tipicamente independentes da aplicação. Geralmente são necessárias para cumprir alguma responsabilidade do software, e freqüentemente correspondem a entidades de banco de dados.

Classes de controle coordenam o fluxo de um caso de uso complexo, encapsulando lógica que não se enquadra naturalmente nas responsabilidades das entidades. São tipicamente dependentes de aplicação.

Classes de fronteira tratam da comunicação com o ambiente do software, modelam as interfaces do software com os usuários e outros softwares, e surgem tipicamente de cada par ator-caso de uso.

2.2.2.4 Atributos

O atributo é a menor unidade que em si possui significância própria e inter-relacionada com o conceito lógico da classe a qual pertence. Apresenta um princípio de atomicidade, ou seja, do armazenamento de um valor simples em uma célula (FURLAN, 1998, p. 102). Uma classe pode ter qualquer número de atributos ou até mesmo nenhum atributo. Em um

determinado momento um objeto de classe terá valores específicos para cada um dos atributos de sua classe (BOOCH, 2000, p. 50).

No modelo de classes de domínio, os atributos, quando representados, o são somente pelo nome. No entanto a sintaxe completa da UML para definição de um atributo é bem mais detalhada (BEZERRA, 2003, p. 167).

O Quadro 1 apresenta a sintaxe de especificação para assinatura de um atributo.

[/] ***visibilidade nome : tipo* = *valor inicial***

Fonte: adaptado de Bezerra (2003, p. 167).

Quadro 1 – Especificação de atributo

O elemento nome da sintaxe do atributo corresponde ao nome do atributo. Na verdade, somente esse elemento é obrigatório na sintaxe de declaração de um atributo (BEZERRA, 2003, p. 168). O elemento tipo especifica o tipo de atributo. Esse elemento é dependente da linguagem de programação utilizada na qual a classe deve ser implementada. O tipo definido para um atributo pode ser definido através da utilização de um tipo primitivo da linguagem de programação a ser utilizada na implementação. Um valor inicial também pode ser declarado para o atributo. Desta forma, sempre que um objeto de uma classe é instanciado, o valor inicial declarado é automaticamente definido para o atributo. Assim como o tipo, o valor inicial de um atributo também é dependente da linguagem de programação (BEZERRA, 2003, p. 169).

Os atributos podem ser marcados por quatro tipos de visibilidade: *public*, *private*, *protected* ou *package*. A visibilidade *public* possui como notação o sinal de adição (+), permitindo que o atributo seja acessado por todos os membros internos e externos à classe. A visibilidade *private* possui como notação o sinal de subtração (-), permitindo que o atributo seja acessado somente pelos membros internos da classe. A visibilidade *protected* possui como notação um sinal de cerquilha (#), permitindo que o atributo seja acessado somente pelos membros da classe e de suas subclasses. A visibilidade *package* possui como notação o sinal de til (~), permitindo que o atributo seja acessado por todos os membros internos e externos à classe que estejam no mesmo pacote (OBJECT MANAGEMENT GROUP, 2004).

Os atributos possuem quatro modificadores: *static*, *final*, *transient*, e *volatile*. O modificador *static* indica que o atributo marcado com o mesmo permanecerá na classe,

enquanto existirem objetos instanciados daquele tipo. O atributo marcado com modificador *final* não pode ter o seu valor de inicialização alterado, isso quer dizer que dado um valor a este objeto, este não sofrerá mais mudanças. O modificador *transient* quando marcado para o atributo indica que o mesmo não poderá ser salvo e nem restaurado pelo mecanismo de serialização. O modificador *volatile* quando marcado para o atributo indica que o mesmo terá seu valor alterado assincronamente por segmentos que estejam sendo executados ao mesmo tempo (MEDEIROS, 2004, p. 90).

Pode se definir um atributo derivado em uma classe. Um atributo é derivado quando o seu valor pode ser obtido a partir do valor de outros atributos. Por exemplo, o valor da idade de uma pessoa pode ser obtido a partir de sua data de nascimento. Um atributo derivado é representado por uma barra inclinada (/) à esquerda (BEZERRA, 2003, p. 169).

2.2.2.5 Operações

Uma operação é a implementação de um serviço que pode ser solicitado por algum objeto da classe para modificar o comportamento. Em outras palavras, uma operação é uma abstração de algo que pode ser feito com um objeto e que é compartilhado por todos os objetos desta classe. Uma classe pode ter qualquer número de operações ou até não ter nenhuma operação. Muitas vezes, mas nem sempre, a chamada a uma operação em um determinado objeto altera os dados ou o estado do objeto (BOOCH, 2000, p. 51).

O Quadro 2 apresenta a sintaxe de especificação para a assinatura de uma operação.

visibilidade nome(parâmetros) : tipo-retorno {propriedades}

Fonte: adaptado de Bezerra (2003, p. 171).

Quadro 2 – Especificação de operação

O elemento nome corresponde ao nome dado a operação. Recomenda-se atribuir um nome a uma operação de forma que este nome lembre o resultado da operação e tenha a forma geral <verbo> + <complemento> (BEZERRA, 2003, p. 171).

As operações têm visibilidade e a sua assinalação é pertinente, porque uma operação pode ser chamada para resolver situações específicas de uma classe ou pode existir para permitir que outras classes acessem seus atributos. As operações podem ser marcadas por quatro tipos de visibilidade: *public*, *private*, *protected* ou *package* (MEDEIROS, 2004, p. 92).

Uma operação *public* é aquela que pode ser acessada por operações na própria classe onde ela foi declarada e por operações de outras classes, a notação para este tipo de visibilidade é um sinal de adição (+), logo a frente do nome da operação. Uma operação *private* é aquela que pode ser acessada somente na classe em que foi declarada, a notação para este tipo de visibilidade é um sinal de subtração (-), logo a frente do nome da operação. Uma operação *protected* é acessível somente pelas classes que estejam participando da sua estrutura de herança, a notação para este tipo de visibilidade é um sinal de cerquilha (#), logo a frente do nome da operação. Uma operação *package* é acessível somente pelas classes que estejam no mesmo pacote onde a classe que a declarou está, a notação para este tipo de visibilidade é um sinal de til (~), logo a frente do nome da operação (OBJECT MANAGEMENT GROUP, 2004).

As operações possuem cinco modificadores: *static, abstract, final, root e query*. Uma operação marcada como *static* permanecerá na classe, enquanto existirem vários objetos instanciados daquele tipo. Este tipo de operação somente pode ser invocado se, primeiro, referenciarmos a classe e não o objeto que está instanciado. Os atributos utilizados por uma operação *static* também devem estar assinalados com este modificador; a notação para este tipo de modificador é um sublinhado marcando a operação. Uma operação marcada com o modificador *abstract* não tem implementação, mas apenas o protótipo de como ela deve ser. A notação para este tipo de modificador é o nome da operação escrito em itálico. Uma operação marcada como *final* não poderá ser alterada por outra classe, logo uma operação *final* não pode ser marcada também pelo modificador *abstract*. Uma operação marcada com o modificador *root* indica que esta operação está em seu nível mais alto, não existindo nenhum outro nível acima. Uma operação marcada como *query* é por *default* ajustado para *false* indicando que a execução da referida operação poderá alterar o estado da instância (MEDEIROS, 2004, p. 94).

O elemento tipo de retorno representa o tipo de dados do valor retornado por uma operação. Esse tipo depende da linguagem de programação (BEZERRA, 2003, p. 173). Ele determina a semântica de concorrência para chamadas passivas a uma operação. As opções são *sequential, guarded (synchronized) e concurrent*. Na primeira, as chamadas são feitas uma a uma; na segunda, múltiplas chamadas podem ser feitas, porém apenas uma estará ativa; e na terceira, múltiplas chamadas podem ser feitas e ativadas.

2.2.2.6 Parâmetros

Conforme Bezerra (2003, p. 172) os parâmetros de uma operação correspondem a informações que a mesma recebe quando é executada. Normalmente, essas informações são fornecidas pelo objeto remetente da mensagem que requisita a execução da operação no objeto receptor. Uma operação pode ter zero ou mais parâmetros. Os parâmetros são separados por vírgulas.

O Quadro 3 apresenta a sintaxe de especificação para a assinatura de um parâmetro.

direção nome-parâmetro: tipo-parâmetro

Fonte: adaptado de Bezerra (2003, p. 172).

Quadro 3 – Especificação de parâmetro

O elemento direção serve para definir se o parâmetro pode ou não ser modificado pela operação. Através deste elemento, o modelador pode definir se o parâmetro é de entrada (*in*), saída (*out*) ou ambos (*inout*) (BEZERRA, 2003, p. 172). Parâmetros marcados como *in* não podem ser modificados. Os parâmetros marcados como *out* podem ser modificados para comunicar informações a quem fez a chamada. E os parâmetros marcados como *inout* poderão ser modificados (BOOCH, 2000, p. 129).

O elemento nome-parâmetro corresponde ao nome do parâmetro. Cada parâmetro deve ter um nome único dentro da assinatura da operação. O elemento tipo-parâmetro corresponde ao tipo do parâmetro e é dependente da linguagem de programação (BEZERRA, 2003, p. 172).

2.2.2.7 Relacionamentos

Um relacionamento é uma conexão entre itens. Em uma modelagem orientada a objetos, os três relacionamentos mais importantes são as dependências, as generalizações e as associações. Um relacionamento é representado graficamente como um caminho, com tipos diferentes de linhas para diferenciar os tipos de relacionamentos (BOOCH, 2000, p. 62).

O relacionamento de dependência indica que uma classe depende dos serviços fornecidos por uma outra classe. O relacionamento de dependência é mais utilizado para denotar características de implementação, os tipos de dependências possíveis são por atributo, por variável global, por variável local e por parâmetro (BEZERRA, 2003, p. 176).

Generalização é o nome dado entre dois elementos, um mais geral (pai) e um mais específico (filho). O mais específico é completamente consistente com o mais geral e adiciona informações ao mesmo (MEDEIROS, 2004, p. 78). Para entender a semântica de uma generalização, é preciso lembrar que uma classe representa um conjunto de objetos que partilham um conjunto comum de propriedades (Atributos, Operações, Associações). O principal ganho que se pode obter com a utilização de generalização na modelagem de classes é tornar o modelo mais simples e realizar o reuso de conceitos definidos previamente para definir novos conceitos (BEZERRA 2003, p. 191).

Uma associação é um relacionamento estrutural que especifica objetos de um item conectados a objetos de outro item. A partir de uma associação conectando duas classes, se é capaz de navegar do objeto de uma classe até o objeto de outra classe e vice-versa. É inteiramente válido ter as duas extremidades do círculo de uma associação retornando a mesma classe. Isso significa que, a partir de um objeto da classe, se pode criar vínculos com outros objetos da mesma classe. Uma associação que estabelece uma conexão exata a duas classes é chamada uma associação binária (BOOCH, 2000, p. 63).

Além da sua forma básica, existem quatro tipos de aprimoramentos a serem aplicados às associações, sendo eles o nome, o papel, a multiplicidade e a agregação (BOOCH, 2000, p. 64).

O nome pode ser utilizado para descrever a natureza do relacionamento. Portanto, não havendo ambigüidade acerca de seu significado, se pode atribuir uma direção para o nome, fornecendo um triângulo de orientação que aponta a direção como o nome deverá ser lido (BOOCH, 2000, p. 64). Quando uma classe participa de uma associação, ela tem um papel específico a executar neste relacionamento; o papel é apenas a face que a classe próxima a uma das extremidades apresenta à classe encontrada na outra extremidade da associação (BOOCH, 2000, p. 65).

As associações permitem representar a informação dos limites inferior e superior da quantidade de objetos aos quais um outro objeto pode estar associado. Esse limites são chamados multiplicidades na terminologia da UML. Cada associação em um diagrama de classes possui duas multiplicidades, uma em cada extremo da linha de associação (BEZERRA, 2003, p. 99).

Os símbolos possíveis para representar uma multiplicidade estão descritos na Tabela 1.

Tabela 1 – Simbologia para representar multiplicidades

Nome	**Simbologia**
Apenas Um	1
Zero ou Muitos	0..*
Um ou Muitos	1..*
Zero ou Um	0..1
Intervalo Específico	$1_i..1_s$

Fonte: adaptado de Bezerra (2003, p.100).

Existem infinitas possibilidades de associação entre objetos (todas as combinações possíveis entre os números inteiros). No entanto, essas associações podem ser agrupadas em três grandes tipos: “muitos para muitos”, “um para muitos” e "um para um”. Denomina-se conectividade o tipo de associação entre duas classes. O tipo de conectividade da associação entre duas classes depende dos símbolos de multiplicidade que são utilizados na associação (BEZERRA, 2003, p. 101). As correspondências entre os tipos de conectividade e os símbolos de multiplicidades estão descritos na Tabela 2 .

Tabela 2 – Conectividades X Multiplicidades

Conectividade	**Multiplicidade de um extremo**	**Multiplicidade do outro extremo**
Um para um	0..1 ou 1	0..1 ou 1
Um para muitos	0..1 ou 1	* ou 1..* ou 0..*
Muitos para muitos	* ou 1..* ou 0..*	* ou 1..* ou 0..*

Fonte: adaptado de Bezerra (2003, p.101).

Uma característica importante de uma associação está relacionada a necessidade ou não da existência dessa associação entre objetos. Essa característica é denominada participação. A participação pode ser obrigatória ou parcial. Se o valor mínimo da multiplicidade de uma associação é igual a 1 (um), significa que a participação é obrigatória. Caso contrário, a participação é opcional (BEZERRA, 2003, p. 102).

Estruturas todo-parte são representadas por meio de agregação, que podem ser de dois tipos: agregação e agregação por composição (MEDEIROS, 2004, p. 96).

Uma agregação pode ocorrer somente entre duas classes. Este tipo de relacionamento ou associação, conforme a UML, é identificável a partir da notação de uma linha sólida entre duas classes. A classe denominada todo-agregado recebe um diamante vazio, enquanto a outra ponta da linha é ligada à classe denominada parte-constituinte. Ambas as classes podem “viver” de forma independente, ou seja, não existe uma ligação forte entre as duas. Objetos da

parte constituinte ou da parte agregado são independentes em termos de vida, porém ambas são partes de um único todo (MEDEIROS, 2004, p. 96).

Uma agregação de composição ocorre quando temos uma situação semelhante a da agregação entre duas classes, porém os objetos da classe parte não podem viver quando o todo é destruído. Este tipo de relacionamento ou associação, conforme a UML, é identificável pela notação de uma linha sólida entre duas classes, a classe denominada todo-composta recebe um diamante preenchido, enquanto a outra ponta da linha é ligada a classe denominada parte-componente. Ambas as classes “vivem“ unidas de forma dependente, ou seja, existe uma ligação forte entre as duas. Os objetos da classe parte são dependentes, em termos de vida, da classe todo, ambas são partes de um único todo (MEDEIROS, 2004, p. 98).

2.3 XML (*EXTENSIBLE MARKUP LANGUAGE*)

O XML é uma tecnologia derivada do SGML (*Standart Generalized Markup Language*) e permite a troca de dados de forma padronizada em meios eletrônicos, estando sob a responsabilidade técnica da W3C (*Word Wide Web Consortium*) (MEDEIROS, 2004, p. 139).

Anderson (2001, p. 17) conceitua XML como uma abordagem simples para marcar o conteúdo com *tags* para armazenar informações. As *tags* delimitam o conteúdo e a sintaxe do XML, o que permite definir estruturas de complexidade arbitrária. Tudo isto é feito com textos normais, formatos de dados não binários, fazendo com que esta seja uma grande solução para troca de informação entre plataformas. O XML leva tudo isto a um novo nível, pois se pode ampliar o XML arbitrariamente para alcançar novas necessidades e usos. Uma vez que a extensão do mecanismo é padrão, pode-se escrever automaticamente a extensão para qualquer um que leia os dados.

O XML é uma linguagem estruturada porque, se existe uma *tag* de abertura, deve haver uma de fechamento, e também porque é dado um significado para cada *tag*. O *Parse* XML é um verificador da sintaxe correta utilizada no arquivo XML. Ele sabe o que está certo e o que está errado, porque existe uma seção no arquivo XML, ou mesmo um arquivo a parte, com extensão DTD (*Document Type Definition)*, que informa qual é a formatação das *tags* (MEDEIROS, 2004, p. 139).

2.4 FERRAMENTAS CASE (*COMPUTER AIDED SOFTWARE ENGINEERING*)

Bezerra (2003, p. 39) define ferramentas CASE como sistemas de software que são utilizados para dar suporte ao ciclo de vida do desenvolvimento, Martin (1994, p. 8) complementa esta definição como softwares que empregam as representações gráficas na tela para ajudar a planejar, analisar, projetar e gerar software.

Furlan (1998, p. 65) alerta que é essencial avaliar completamente os benefícios da ferramenta e suas limitações de maneira a evitar problemas posteriores no ciclo de desenvolvimento. As ferramentas CASE podem ser categorizadas como CASE Integrado (ou I-CASE), CASE Fragmentado e CASE-IE (*Information Engineering*) (MARTIN, 1994, p. 330).

O CASE Integrado refere-se a um pacote de ferramentas CASE que suporta todos os estágios do ciclo de vida de desenvolvimento, inclusive o da geração de código, com um único repositório, logicamente consistente. No CASE Integrado, as ferramentas para todos os estágios do desenvolvimento se unem e alimentam um gerador de código (SCHMIDT, 2001, p. 9).

Ao contrário do CASE Integrado, o CASE Fragmentado suporta apenas uma porção do ciclo de vida do software. Esta categoria de CASE inclui as ferramentas de *front-end*, que agrupam a parte de análise, e as ferramentas de *back-end*, que se referem à geração de código (SCHMIDT, 2001, p. 9). Existem ainda os CASE-IE que suportam totalmente a engenharia da informação (SCHMIDT, 2001, p. 10).

2.4.1 XMI (*XML METADATA INTERCHANGE*)

Conforme a OMG (*Object Management Group*), XMI é um formato padrão de intercâmbio de meta dados entre aplicações, repositórios e ferramentas baseado em XML. Entre os benefícios trazidos com a utilização desse padrão estão a facilidade de implementação, por parte das empresas, nos produtos atuais, quebra da barreira entre ferramentas, repositórios e aplicações incompatíveis e o fato de trabalhar com tecnologias que já são padrões da indústria como XML e UML (OBJECT MANAGEMENT GROUP, 2002).

2.4.2 GERADORES DE CÓDIGO

Fisher (1990, p. 10) define que a geração automática do código, integral ou parcial, fornece as seguintes vantagens: redução do tempo de desenvolvimento, pois minimiza a necessidade de codificação manual, e aumento da confiabilidade do código gerado, o qual foi produzido por uma ferramenta depurada e testada, afirmando também que a geração automática de código define a capacidade de gerar automaticamente um software funcional ou compilável diretamente de uma especificação de projeto.

Martin (1994, p. 327) expõe o seguinte prognóstico: "Os geradores de código deverão tornar-se o principal meio de se criar aplicativos baseados em objetos, com um repositório CASE e classes reaproveitáveis. Com as ferramentas CASE-OO (*Object Oriented*), a ênfase maior da construção de softwares será sobre a modelagem e sobre o projeto, e não sobre a programação. Dessa forma, as metodologias de análise baseadas em objetos não deveriam estar atreladas a linguagens baseadas em objetos e seus recursos. Uma vez que a tecnologia de desenvolvimento tem mudado tão rapidamente, deve-se analisar os softwares, independentemente dos recursos de programação utilizados".

As ferramentas I-CASE mais potentes permitem que o projeto seja sintetizado a partir de construções de alto nível contidas no repositório. O projetista monta o projeto e cria sua lógica detalhada. Parte do projeto pode ser gerada a partir de declarações comportamentais e estruturais de alto nível (SCHMIDT, 2001, p. 11).

2.5 INTERNACIONALIZAÇÃO

A internacionalização é a modificação ou criação de produtos com o objetivo de facilitar o uso dos mesmos em diferentes países e idiomas. A internacionalização de software faz com que os mesmos aceitem diversas codificações de dados necessárias para o processamento de informações multilíngües, bem como formatos de moedas, calendários, métodos de entrada e outras adaptações necessárias para atender os requisitos culturais (LIONBRIDGE, 2005).

Conforme Araújo (2003, p. 44) para um software ser competitivo no mercado mundial, o mesmo precisa se comunicar com o usuário final na sua língua nativa, precisando também estar baseado nas suas convenções locais. Para muitas organizações, a internacionalização pode ser vista como uma importante janela para o mercado externo, na medida em que o

software poderá ser preparado desde as fases iniciais do ciclo de vida para rodar em qualquer lugar do mundo sem que seja necessário mudar o código ou recompilá-lo para diferentes línguas e convenções locais (ARAÚJO, 2003, p. 45).

2.6 TRABALHOS CORRELATOS

Este tópico apresenta alguns trabalhos correlatos, identificando suas principais características, autores e a contribuição dos mesmos para o presente trabalho.

Ballmann (2000) apresenta como trabalho de conclusão de curso um protótipo de ferramenta CASE (*Computer Aided Software Engineering*) para geração de código C++ e diagrama de classes. O objetivo deste trabalho foi desenvolver uma ferramenta para criação do diagrama de classes, conforme o padrão da UML (*Unified Modeling Language*), podendo gerar código fonte em C++ a partir de um diagrama de classes da ferramenta CASE Rational Rose e permitir a engenharia reversa, transformando um código fonte C++ em um diagrama de classes.

Kramel (2000) apresenta como trabalho de conclusão de curso um protótipo de software para geração de código CDL (*Class Definition Language*) através do repositório da ferramenta CASE System Architect. O objetivo deste trabalho foi desenvolver um protótipo de software de geração de código CDL para o banco de dados Cachê a partir do repositório da ferramenta CASE System Architect.

Schmidt (2001) apresenta como trabalho de conclusão de curso uma ferramenta de auxílio ao processo de desenvolvimento de software integrando tecnologias otimizadoras. O objetivo deste trabalho consiste no desenvolvimento de uma ferramenta CASE que possibilitasse a geração de código fonte a partir das definições contidas no repositório de dados da ferramenta de modelagem ERwin, complementadas pelo repositório de dados da ferramenta desenvolvida pelo trabalho.

Silveira (2003) apresenta como trabalho de conclusão de curso uma ferramenta para geração automática de cadastros e consultas para linguagem ASP (*Active Server Pages*) baseado em banco de dados. O objetivo deste trabalho foi desenvolver uma ferramenta de auxílio ao programador para criação automática de um sistema para cadastros e consultas na plataforma *web*, utilizando linguagem ASP, baseado na estrutura de dados e relacionamentos do banco de dados.

Wehrmeister (2001) apresenta como trabalho de conclusão de curso um protótipo de software para geração de código fonte a partir do repositório da ferramenta CASE System Architect. O objetivo deste trabalho foi desenvolver um protótipo de software para a geração de código fonte em diversas linguagens de programação, desde que sua especificação esteja em um arquivo texto utilizando a notação BNF (*Backus-Naur Form*) estendida, a partir do repositório da ferramenta CASE System Architect.

Os trabalhos correlatos assim como o presente trabalho têm como finalidade a construção de uma ferramenta para geração de código. Nota-se que apesar de terem a mesma finalidade, os mesmos utilizam diferentes processos e tecnologias. O presente trabalho procurou utilizar algumas tecnologias e conceitos identificados nos trabalhos correlatos, exemplos disto é a utilização do diagrama de classes e a preocupação em utilizar um mesmo repositório de dados na geração de código para diferentes arquiteturas.

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo aborda o desenvolvimento do trabalho e está dividido em quatro tópicos, o primeiro tópico descreve os requisitos atendidos pela ferramenta. O segundo tópico refere-se à especificação da ferramenta, onde são apresentados alguns diagramas UML (*Unified Modeling Language*) visando uma melhor compreensão do mesmo. O terceiro tópico refere-se a implementação da ferramenta, onde está descrito seu funcionamento, tecnologias utilizadas e operacionalidade da implementação. No último tópico são apresentados os resultados obtidos.

3.1 REQUISITOS DA FERRAMENTA

Os requisitos da ferramenta compreendem o levantamento das funcionalidades e/ou necessidades dos usuários para criação da ferramenta. Estes requisitos são resultados de estudos realizados através do uso das principais ferramentas CASE (*Computer Aided Software Engineering*) existentes no mercado, trabalhos correlatos e das tecnologias utilizadas. A Tabela 3 apresenta os requisitos funcionais identificados para a ferramenta criada.

Tabela 3 – Requisitos funcionais

Requisitos funcionais
RF01: Manter cadastros de projetos, arquiteturas, idiomas, tipos de dado, grupos de propriedades e das especializações dos elementos do diagrama de classes
RF02: Importar diagramas de classes gerados por ferramentas CASE através de artefatos XMI (*XML Metadata Interchange)*
RF03: Permitir ao usuário a especialização de tipos de dado, pacotes, interfaces, classes, atributos, operações, parâmetros e relacionamentos por arquitetura e idioma
RF04: Fornecer ao usuário a opção para geração de código por pacote e classes
RF05: Disponibilizar aos usuários uma interface padrão para acoplamento de *plugins* que contenham a lógica de geração de código

O Tabela 4 apresenta os requisitos não funcionais identificados para a ferramenta criada.

Tabela 4 – Requisitos não funcionais

Requisitos não funcionais
RNF01: Utilizar artefatos criados com base na especificação XMI versão 1.2 na importação dos diagramas de classes
RNF02: Utilizar a tecnologia .NET para codificação da ferramenta
RNF03: Utilizar XML *(Extensible Markup Language)* e Banco de Dados para persistir os dados

3.2 ESPECIFICAÇÃO

Este tópico descreve a especificação da ferramenta através dos diagramas de casos de uso, diagramas de classes e diagrama de atividades.

3.2.1 DIAGRAMA DE CASOS DE USO

Este tópico demonstra a representação gráfica dos casos de uso da ferramenta, a descrição textual dos mesmos encontra-se no Apêndice A. Para melhor visualização dos casos de uso da ferramenta foram utilizados pacotes. Os pacotes são mecanismos de propósito geral para a organização de elementos em grupos (BOOCH, 2000, p. 168). A visão de pacotes dos casos de uso é representada graficamente pela Figura 3 .

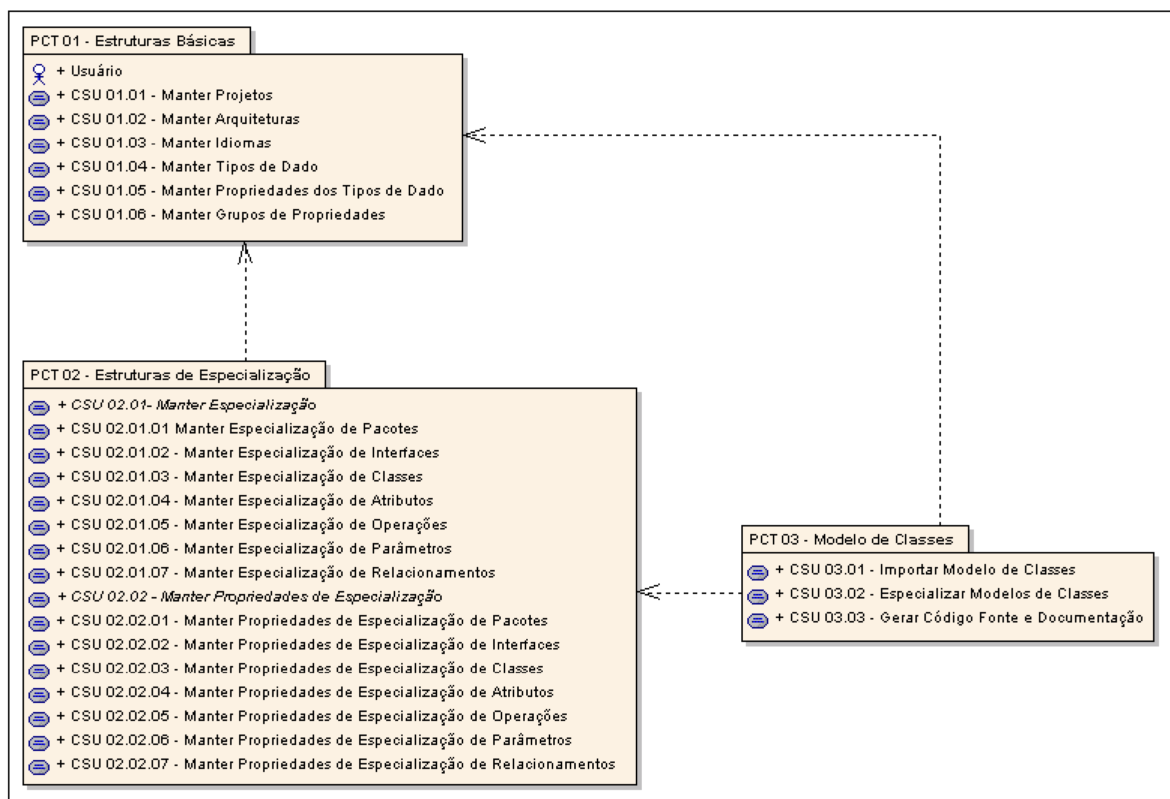


Figura 3 – Pacotes dos casos de uso

A Figura 4 apresenta os casos de uso do pacote 01 (Estruturas Básicas), os casos de uso das estruturas básicas são os primeiros casos de uso com os quais os usuários interagem. Estes casos de uso manipulam as informações básicas utilizadas nos pacotes 02 e 03.

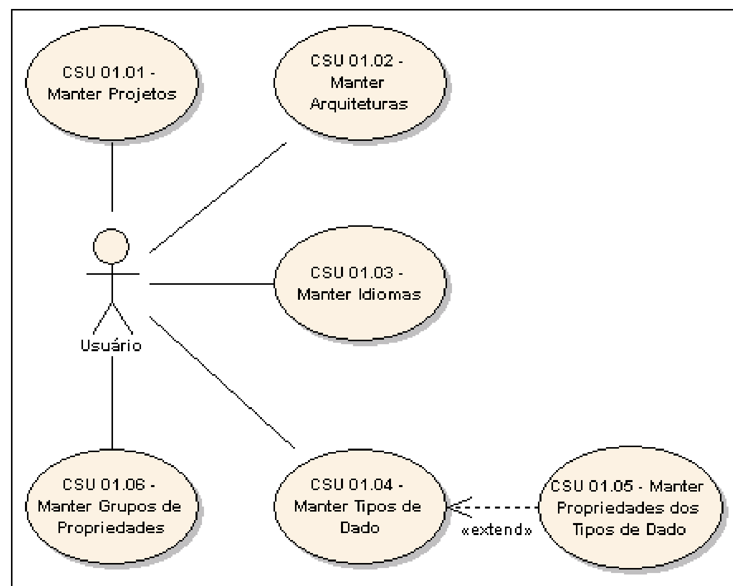


Figura 4 – Casos de uso das estruturas básicas

A Figura 5 apresenta os casos de uso do pacote 02 (Estruturas de Especialização), os casos de uso das estruturas de especialização são idênticos para cada elemento do modelo de classes (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros e Relacionamentos), por este motivo o caso de uso 02.01 (Manter Especialização) e o caso de uso 02.02 (Manter Propriedades de Especialização) são abstratos, servindo somente como referência para os demais casos de uso descendentes, desta forma a descrição textual dos casos de uso de especialização encontrada no Apêndice A refere-se somente aos casos de uso 02.01 e 02.02.

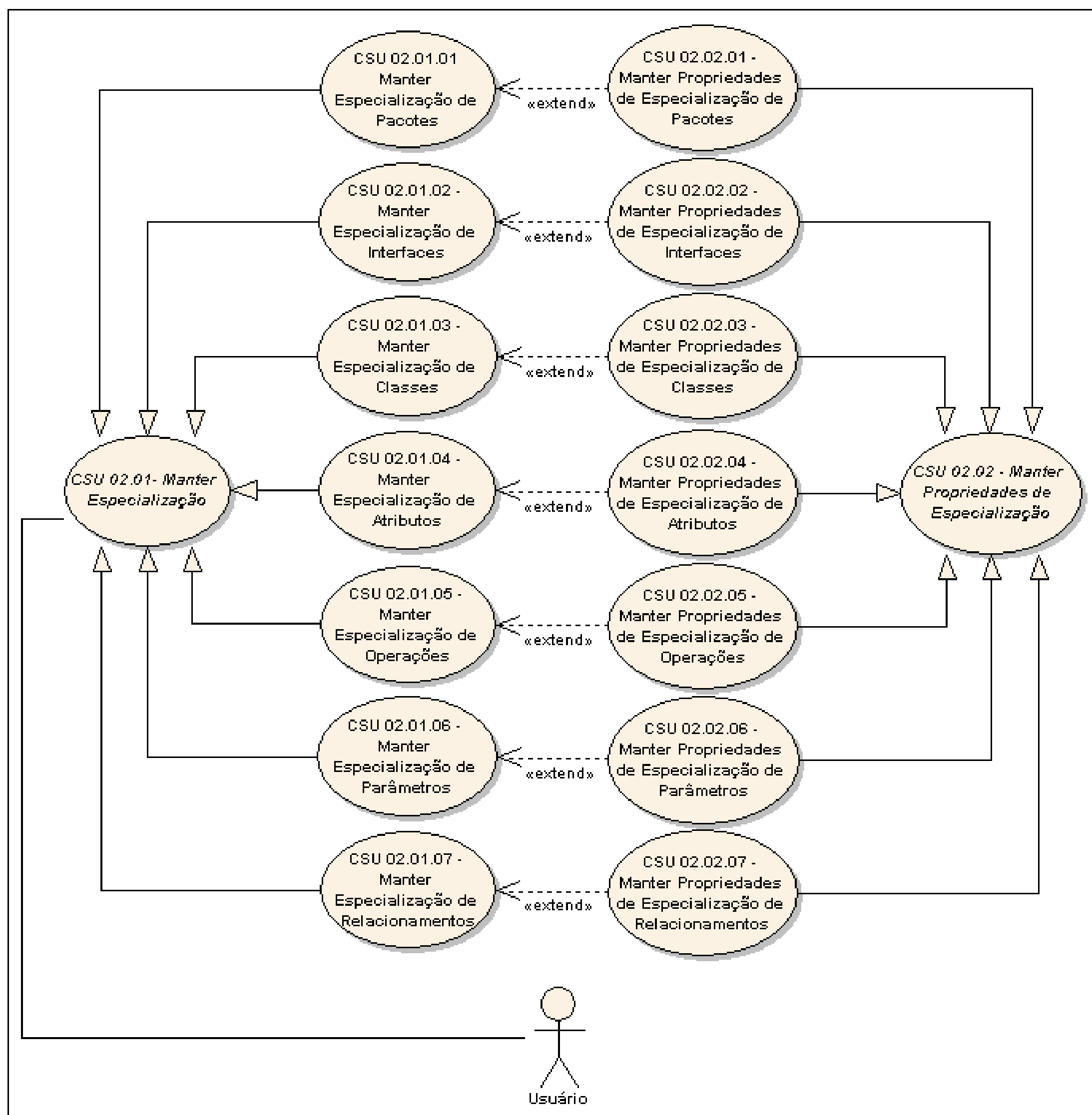


Figura 5 – Casos de uso das estruturas de especialização

A Figura 6 apresenta os casos de uso do pacote 03 (Modelo de Classes), os casos de uso do modelo de classes realizam basicamente a especialização dos modelos de classes e a execução da geração de código.

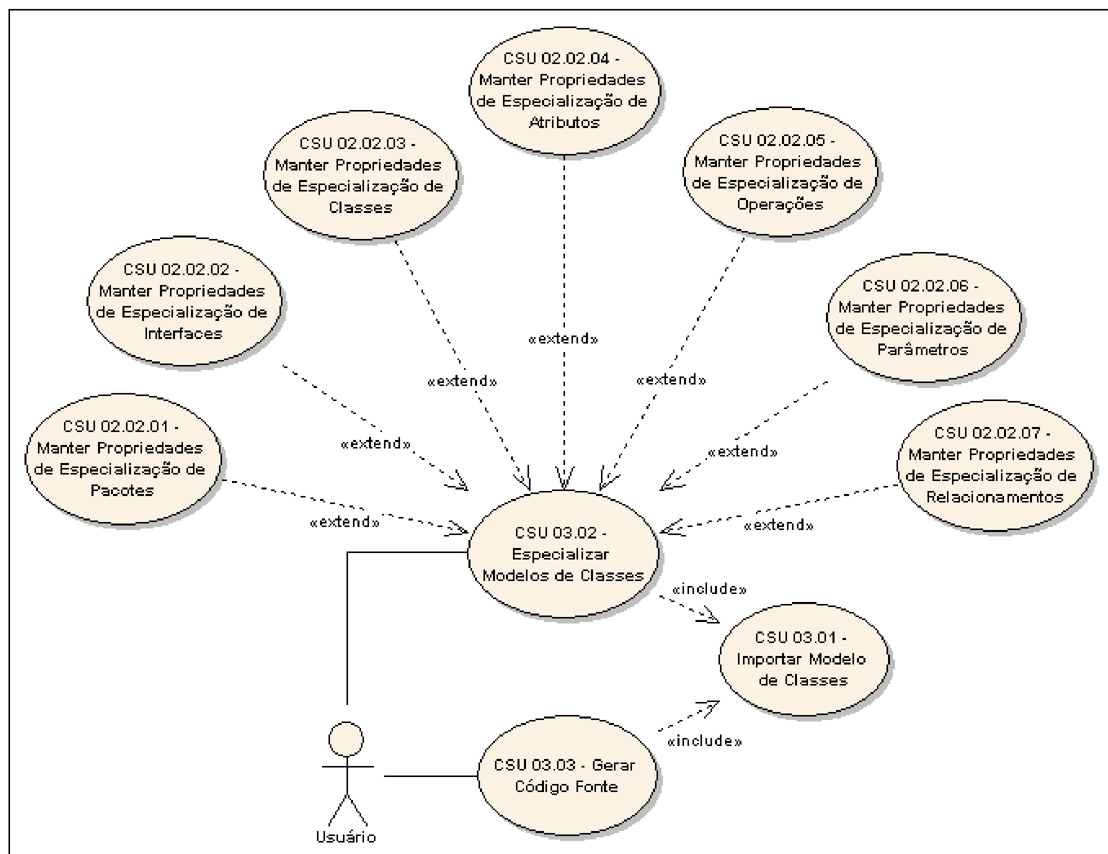


Figura 6 – Casos de uso do modelo de classes

3.2.2 DIAGRAMA DE CLASSES

Este tópico demonstra os diagramas de classes criados durante o desenvolvimento da ferramenta. A Figura 7 apresenta o diagrama de classes de domínio que representa as classes no domínio do negócio em questão, sendo este modelo construído na fase de análise (BEZERRA, 2003, p. 96).

O diagrama de classes de especificação é apresentado a partir da Figura 8. Este diagrama é uma extensão do diagrama de classes de domínio, que recebe a adição de detalhes específicos conforme a solução de software escolhida (BEZERRA, 2003, p. 96). A Figura 8 apresenta os pacotes criados para realizar a divisão da ferramenta em camadas. A divisão de um software em camadas permite que este seja mais portável e modificável, sendo que uma mudança em uma camada mais baixa que não afete sua interface não implicará em mudanças nas camadas mais altas (BEZERRA, 2003, p. 246). As camadas utilizadas para divisão da ferramenta são: camada de persistência apresentada na Figura 9, camada de lógica apresentada na Figura 10 e a camada de apresentação, que é a camada em que ocorre a interação do usuário com a ferramenta, sendo esta apresentada pela Figura 11.

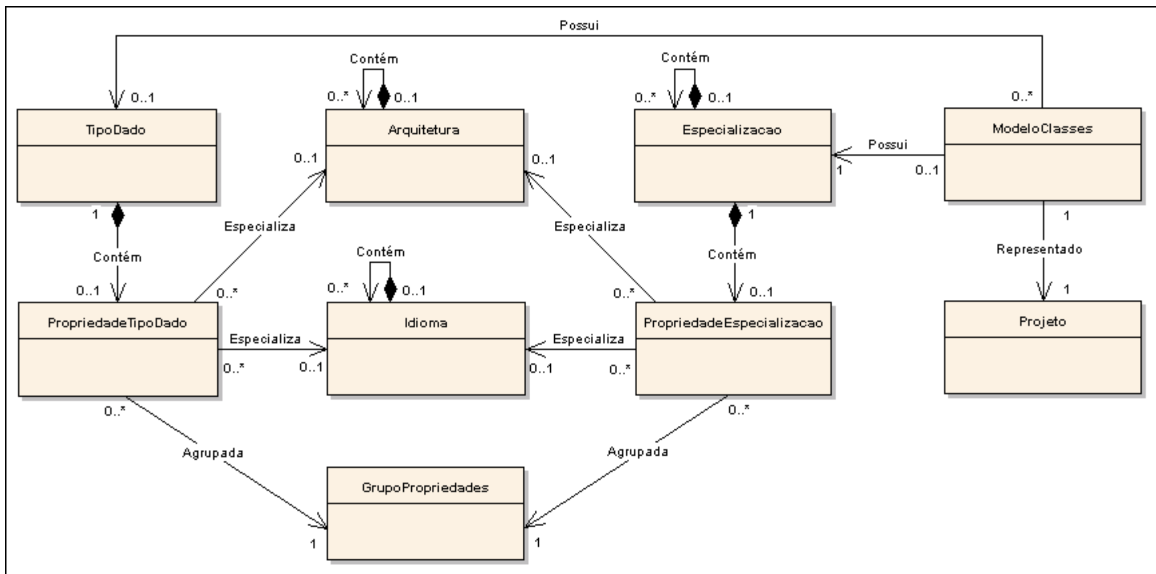


Figura 7 – Diagrama de classes de domínio

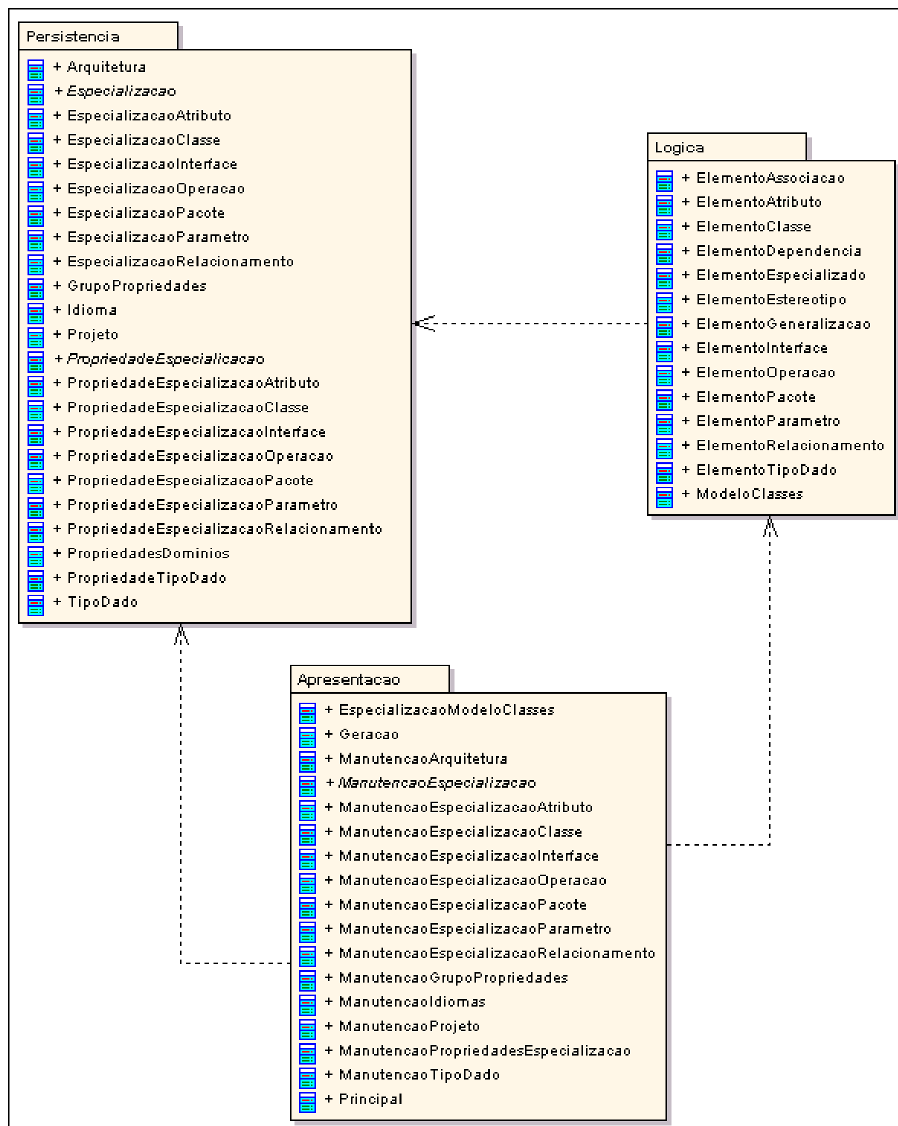


Figura 8 – Pacotes do diagrama de classes de especificação

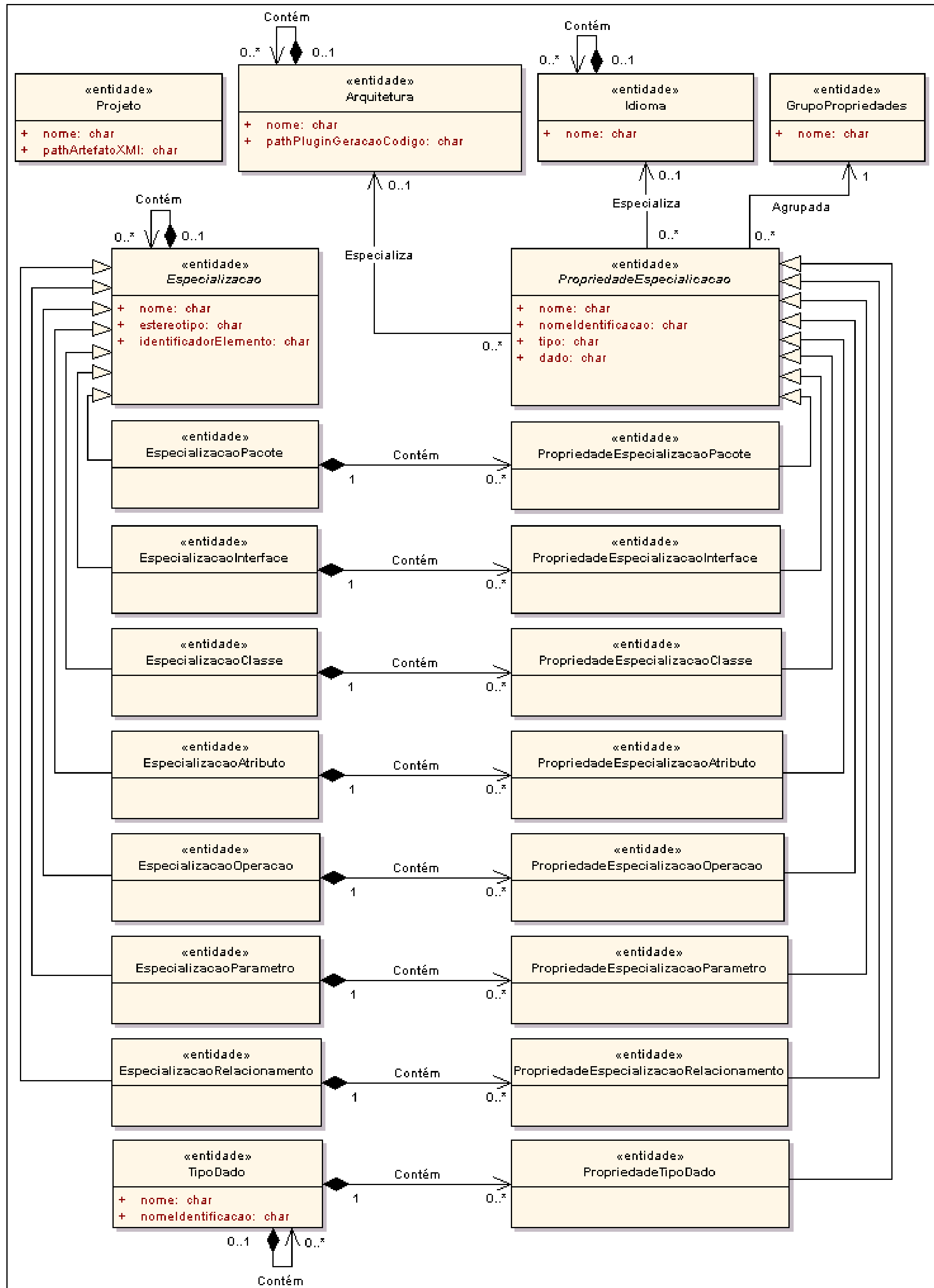


Figura 9 – Diagrama de classes da camada de persistência

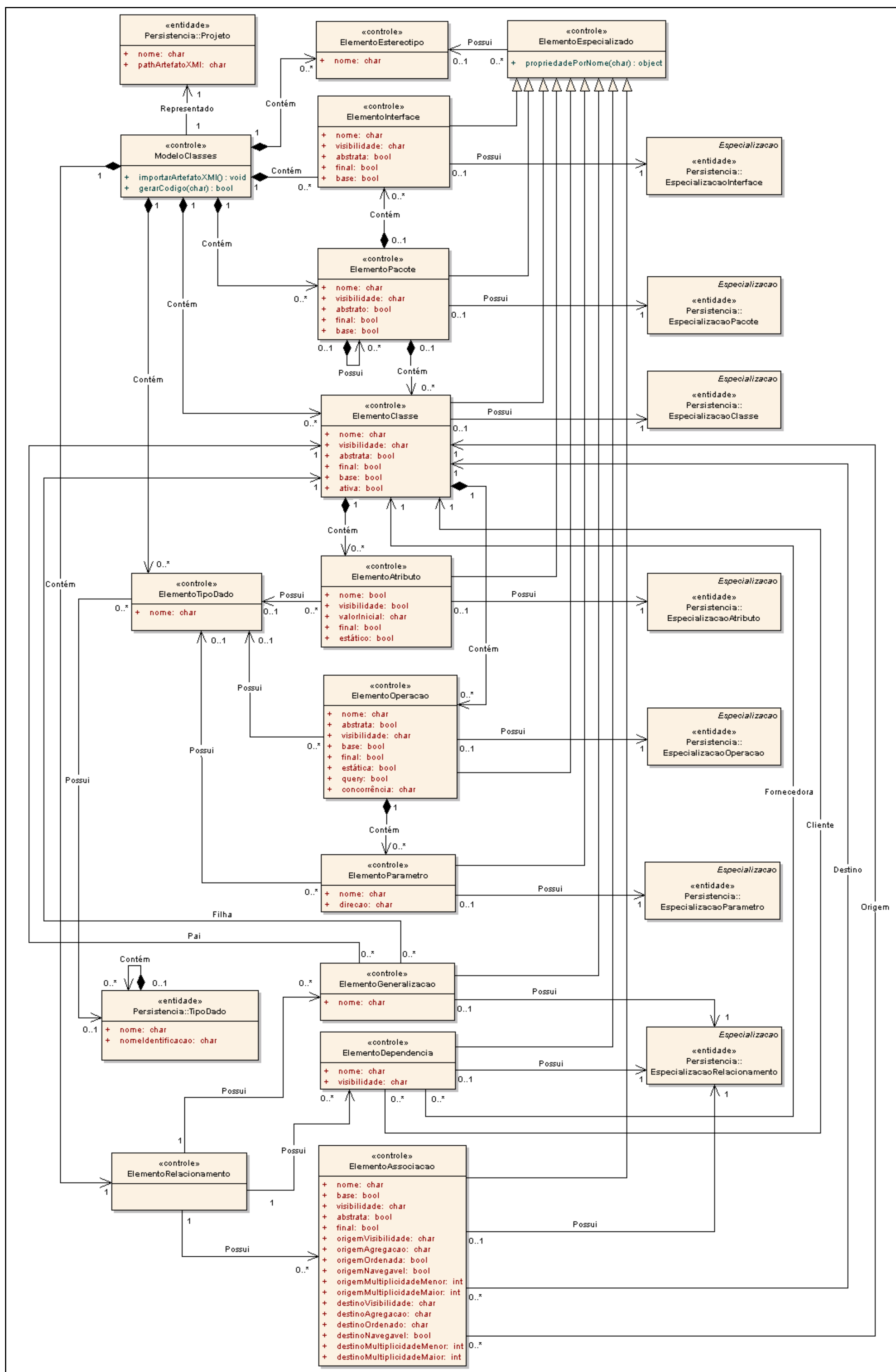


Figura 10 – Diagrama de classes da camada de lógica

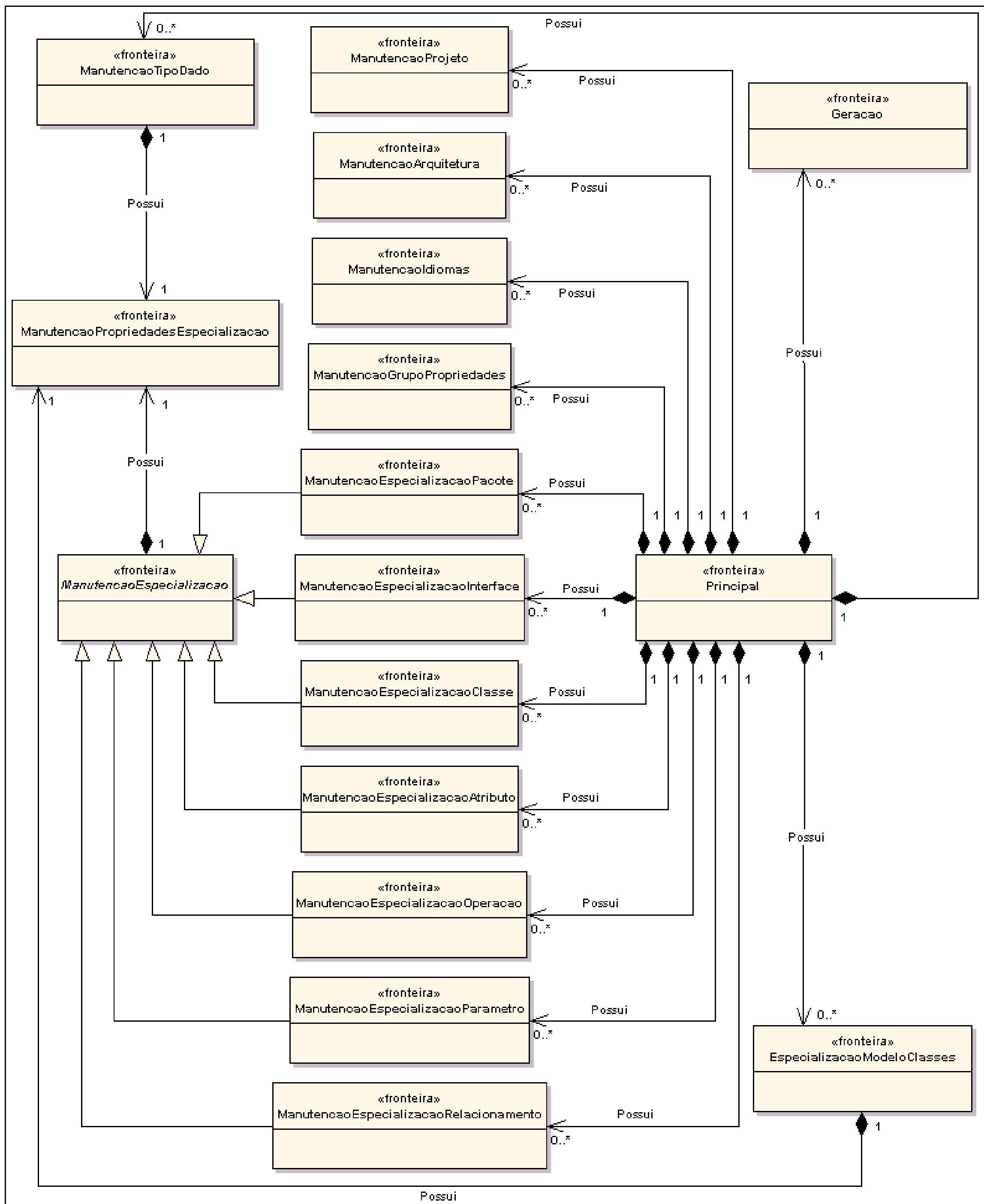


Figura 11 – Diagrama de classes da camada de apresentação

3.2.3 DIAGRAMA DE ATIVIDADES

Este tópico demonstra o diagrama de atividades da ferramenta. Um diagrama de atividades é essencialmente um gráfico de fluxo, mostrando o fluxo de controle de uma atividade para outra (BOOCH, 2000, p. 255). O diagrama de atividades apresentado na Figura 12 demonstra uma visão geral do fluxo das informações e processos necessários para geração de código.

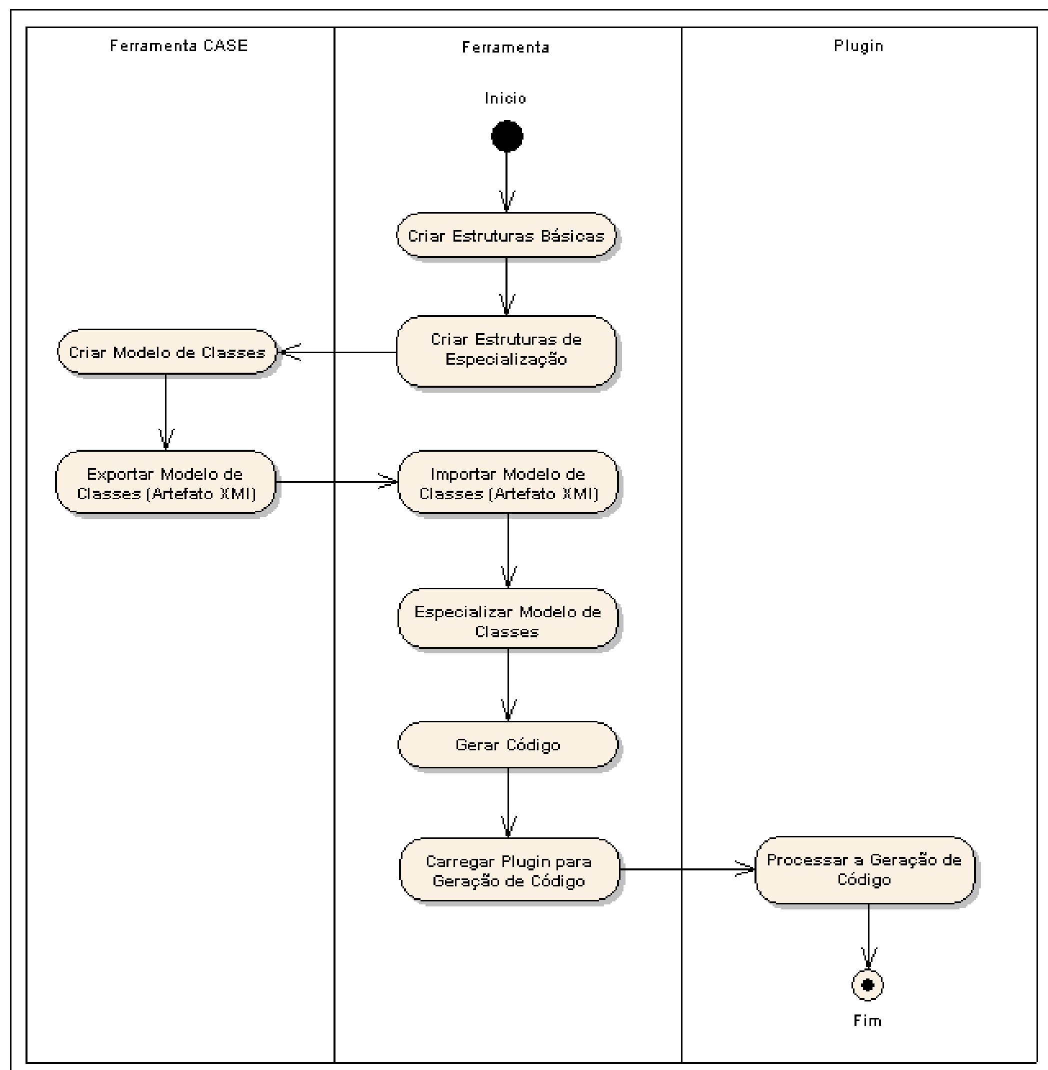


Figura 12 – Diagrama de atividades

3.3 IMPLEMENTAÇÃO

Este tópico contém o detalhamento da implementação da ferramenta, apresentando a estrutura de especialização proposta pelo trabalho, as técnicas e ferramentas utilizadas, um estudo de caso do ponto de vista do usuário que destaca a operacionalidade da implementação e por último um exemplo de geração de código e *plugins*.

3.3.1 ESTRUTURA DE ESPECIALIZAÇÃO

Para uma melhor compreensão dos tópicos seguintes faz-se necessário primeiramente o entendimento da estrutura de especialização proposta pelo trabalho. A estrutura de especialização visa agregar a cada elemento do diagrama de classes (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros e Relacionamentos) novas características ou propriedades.

3.3.1.1 Categorias de especialização e suas propriedades

Este tópico demonstra o objetivo da criação de categorias de especialização e de suas propriedades. Uma categoria de especialização é composta por um conjunto de propriedades e tem o objetivo de manter um agrupamento das mesmas por categoria. Um exemplo de categoria de especialização pode ser apresentado através das diferentes categorias de “classes” propostas pela Engenharia de Software, onde as mesmas podem ser divididas em classes de entidades, classes de controle e classes de fronteira, sendo que cada uma das categorias citadas apresentam características ou propriedades de caráter particular.

Uma propriedade de especialização possui nove tipos de dados. Ao criar uma propriedade de especialização o usuário realiza a escolha do tipo de dado que a propriedade deve armazenar. A Figura 13 apresenta um *menu* com os tipos de dado citados.

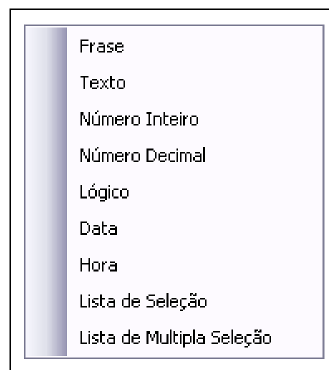


Figura 13 – Tipos de dado das propriedades de especialização

Os tipos de dados apresentados visam oferecer ao desenvolvedor uma maior flexibilidade e organização na criação das propriedades de especialização. Como exemplo pode se citar o tipo de dado “Lógico” que pode ser utilizado na especialização de um determinado “atributo”, indicando se o mesmo é visível ou não, outro exemplo é o tipo de dado “Texto” que pode ser utilizado para documentar os elementos do diagrama de classes.

As propriedades de especialização possuem quatro tipos de visibilidade, cada tipo pode estar relacionado com uma arquitetura e/ou idioma. A Tabela 5 apresenta os tipos de visibilidade juntamente com seus níveis de dependência.

Tabela 5 – Tipos de visibilidade das propriedades de especialização

Tipos de visibilidade	**Dependente de arquitetura**	**Dependente de idioma**
Genérica		
Arquitetura	X	
Idioma		X
Arquitetura e Idioma	X	X

O tipo de visibilidade "Genérica" oferece ao usuário a possibilidade da criação de propriedades que sejam completamente independentes de arquitetura e idioma, a utilização deste tipo de propriedade proporciona uma considerável reutilização das especificações armazenadas no repositório da ferramenta. Um exemplo para este tipo de propriedade pode ser apresentado através de um indicativo que informa se um determinado "atributo" aceita valor nulo ou não, nota-se que seu valor se manterá o mesmo independente da arquitetura ou idioma utilizado.

O tipo de visibilidade "Arquitetura" tem o objetivo de agrupar propriedades que pertençam a uma arquitetura em particular. Um exemplo para este tipo de visibilidade pode ser apresentado através da especialização de uma "classe" de interface para arquitetura Delphi, a mesma possui uma propriedade denominada "HelpContext" não contemplada pelas demais arquiteturas.

O tipo de visibilidade "Idioma" tem o objetivo de agrupar propriedades que pertençam a um determinado idioma em particular. Um exemplo para este tipo de visibilidade pode ser apresentado através da criação de uma descrição ou rótulo para um determinado atributo, havendo desta forma a necessidade de especificá-lo para cada um dos idiomas utilizados.

O tipo de visibilidade "Arquitetura e Idioma" é o menos comum, a criação do mesmo se aplica as propriedades de especialização que pertencem a uma determinada arquitetura e idioma. Um exemplo de aplicação deste tipo de visibilidade é a documentação de uma determinada "operação", caso esta seja implementada em diferentes arquiteturas e necessite ser documentada em vários idiomas.

A estrutura de especialização segue o modelo de herança semelhante ao utilizado pela orientação a objetos. A utilização do modelo de herança permite a criação de categorias de especializações descendentes, sendo que as mesmas herdam todas as propriedades da sua categoria de especialização ancestral.

As propriedades de especialização herdadas de uma categoria de especialização ancestral podem ser sobrescritas para a categoria de especialização descendente através da atribuição de um novo valor para a mesma. A Tabela 6 apresenta as possíveis descendências das propriedades de especialização por tipo de visibilidade.

Tabela 6 – Possíveis descendências por tipo de visibilidade

Tipos de visibilidade ancestrais	**Tipos de visibilidade descendentes**
Genérica	Genérica Arquitetura Idioma Arquitetura e Idioma
Arquitetura	Arquitetura Arquitetura e Idioma
Idioma	Idioma Arquitetura e Idioma
Arquitetura e Idioma	Arquitetura e Idioma

3.3.1.2 Utilização das categorias de especializações

Este tópico tem o objetivo de demonstrar a ligação das categorias de especializações com os elementos dos diagramas de classes importados pela ferramenta. Nota-se através da Figura 9 que as categorias de especializações, assim como cada elemento do digrama de classes, possuem um "estereótipo". Este mecanismo é utilizado para estender o significado de um determinado elemento de um diagrama (BEZERRA, 2003, p. 41).

Ao realizar a importação de um diagrama de classes a ferramenta verifica para cada elemento do diagrama a existência de uma categoria de especialização com estereótipo idêntico ao informado para o elemento na fase de modelagem. Verificada a existência a ferramenta cria uma nova especialização para o item importado, utilizando como ancestral a categoria de especialização encontrada.

Através do caso de uso 03.02 (Especializar Modelo de Classes) o usuário tem a possibilidade de sobrescrever os valores das propriedades para os elementos dos digramas de classes importados, informando desta forma as características particulares de cada um.

3.3.2 TÉCNICAS E FERRAMENTAS UTILIZADAS

A ferramenta implementada utiliza metodologias e técnicas abordadas pela Engenharia de Software, dentre elas pode-se citar a orientação a objetos, a divisão em camadas e a modelagem do mesmo através de diagramas UML (*Unified Modeling Language*).

A orientação a objetos é uma abordagem para desenvolvimento de software que organiza problemas e suas soluções como um conjunto de objetos distintos. Uma vantagem do desenvolvimento orientado a objetos é a consistência da linguagem. Pode-se descrever o problema e sua solução nos mesmos termos: classes, objetos, métodos, atributos e comportamentos (PFLEEGER, 2004, p. 214).

A divisão da ferramenta em camadas foi um dos fatores que possibilitou uma maior organização durante a implementação da mesma. A ferramenta foi dividida em três camadas distintas, sendo elas: a camada de persistência que é encarregada de armazenar e recuperar as informações persistentes, sendo esta a camada mais baixa da ferramenta; a camada de lógica, que é responsável por todas as rotinas lógicas da ferramenta; e a camada de apresentação, que é responsável pela interação dos usuários com os casos de uso da ferramenta, sendo esta a camada mais alta.

A visibilidade entre as camadas tem sentido único, partindo da camada mais alta até a mais baixa, garantindo assim uma maior portabilidade e manutenibilidade da ferramenta. A divisão em camadas da ferramenta pode ser visualizada melhor através da Figura 8.

Os modelos e diagramas foram construídos com a utilização da linguagem UML durante a fase de análise e implementação da ferramenta, os mesmos permitiram uma melhor visualização do problema e das soluções a serem alcançadas, permitindo também uma melhor compreensão para as demais pessoas envolvidas no trabalho.

As ferramentas utilizadas na especificação e implementação são ferramentas conceituadas no mercado de desenvolvimento de software. Estas ferramentas juntamente com seus produtores e versões são apresentadas na Tabela 7, sendo apresentado em seguida a descrição detalhada das ferramentas e a utilização das mesmas na implementação.

Tabela 7 – Ferramentas utilizadas na especificação e implementação

Ferramentas	**Produtores**	**Versões**
.NET Framework SDK	Microsoft	2.0.40607 (Beta)
.NET Custom Library	Ápice Engenharia de Software	1.0 (Beta)
Enterprise Architect	Sparx Systems	4.51
Firebird	Firebird Community	1.5.2.4731
IBOConsole	IBObjects	1.1.11.15
IB DataPump	Clever Components	3.4
Visual Studio 2005	Microsoft	8.0.40607 (Beta)
Visual C# 2005	Microsoft	8.0.40607 (Beta)

A especificação foi realizada através da ferramenta CASE (*Computer Aided Siftware Enginnering*) Enterprise Architect, através desta ferramenta foram construídos os diagramas de atividades, casos de uso e classes.

Para a implementação da ferramenta foi utilizada a plataforma de desenvolvimento Microsoft .NET, sendo realizada a codificação da mesma com linguagem de programação Visual C#. O ambiente de desenvolvimento utilizado foi o Microsoft Visual Studio, apesar de a versão utilizada ser beta, o ambiente se demonstrou muito estável.

Para reduzir o tempo de implementação da ferramenta e garantir uma melhor padronização das classes criadas foi utilizada a biblioteca Ápice .NET Custom Library, que permitiu de forma simples realizar a divisão da ferramenta em camadas e torná-la apta a uma futura conversão de idioma.

Para realizar a persistência dos dados da ferramenta foi construído um banco de dados com a utilização do SGDB (Sistema Gerenciador de Banco de Dados) Firebird. Para construção do mesmo foi utilizada a ferramenta IBOConsole. Esta ferramenta permite a criação e manipulação dos dados contidos em um banco de dados Firebird ou Interbase. Com as constantes mudanças do dicionário de dados durante a fase de implementação, tornou-se necessário o uso de uma ferramenta para recuperação dos dados já cadastrados no banco de dados de teste da ferramenta. Por apresentar grande flexibilidade e por ser de fácil operabilidade, a ferramenta escolhida foi o IB DataPump. Através do uso desta ferramenta torna-se possível a importação dos dados da versão desatualizada do banco de dados para a versão atual.

Além das ferramentas utilizadas para a especificação e implementação, também foram utilizadas algumas tecnologias que facilitaram o desenvolvimento do trabalho e agregaram

uma maior padronização ao mesmo. Estas tecnologias juntamente com os órgãos responsáveis pelas mesmas e suas versões são apresentadas na Tabela 8.

Tabela 8 – Tecnologias utilizadas na especificação e implementação

Tecnologias	**Órgãos Responsáveis**	**Versões**
UML (*Unified Modeling Language*)	OMG (*Object Management Group*)	2.0
XML (*Extensible Markup Language*)	OMG (*Object Management Group*)	1.0
XMI (*XML Metadata Interchange)*	OMG (*Object Management Group*)	1.2

A linguagem UML foi utilizada em conjunto com a ferramenta CASE Enterprise Architect para a especificação da ferramenta, onde foram construídos os modelos e diagramas da mesma.

A tecnologia XML foi utilizada nas rotinas de persistência da ferramenta, em conjunto com o banco de dados. E pelos artefatos de configuração e de idioma. Um exemplo da utilização da tecnologia XML em conjunto com o banco de dados é o armazenamento das informações das propriedades de especialização, conforme relatado no caso de uso 2.2 encontrado no Apêndice A.

As propriedades de especialização podem ser de diferentes tipos, sendo assim o uso da tecnologia XML permitiu a persistência de cada tipo em um único registro de propriedade, conforme demonstrado na Figura 14, que apresenta três propriedades de especialização de tipos diferentes. A propriedade Papel que é do tipo texto, a propriedade Altura que é do tipo número inteiro e a propriedade Sigla que é do tipo frase.

IN_ID	ST_NAME	SL_TYPE	TX_DATA
1	Papel	tx	<tps vl="Controlar a Lógica" />
6	Altura	in	<tps vl="21" />
9	Sigla	st	<tps vl="PRJ" />

Figura 14 – Exemplo de registros de propriedades de especialização

A tecnologia XMI foi utilizada pela ferramenta para a importação dos diagramas de classes gerados por ferramentas CASE.

No Quadro 4 pode-se observar o trecho de um artefato XMI gerado através da ferramenta CASE Enterprise Architect. Este artefato foi utilizado para realização de testes de importação durante a implementação da ferramenta.

```
<?xml version="1.0" encoding="windows-1252" ?>
- <XMI xmi.version="1.2" xmlns:UML="org.omg.xmi.namespace.UML" timestamp="2005-05-16 15:17:04">
  - <XMI.header>
    - <XMI.documentation>
        <XMI.exporter>Enterprise Architect</XMI.exporter>
        <XMI.exporterVersion>4.1</XMI.exporterVersion>
      </XMI.documentation>
      <XMI.metamodel XMI.name="UML" XMI.version="1.4" />
    </XMI.header>
  - <XMI.content>
    - <UML:Model name="Visão de Classes" xmi.id="MX_EAID_AE204C59_20B7_46a5_84BC_61364E8E32C9" isRoot="true"
        isLeaf="false" isAbstract="false" isSpecification="false" visibility="public">
      - <UML:Namespace.ownedElement>
        - <UML:Package name="Basico" xmi.id="EAPK_450EA2BE_FC02_44ee_BB63_1D7C8629024F" isRoot="true" isLeaf="false"
            isAbstract="false" visibility="public">
          - <UML:ModelElement.stereotype>
              <UML:Stereotype xmi.idref="EAID_7EEA4DAC_7DDD_458a_A9C2_0E354A5CD3E1" />
            </UML:ModelElement.stereotype>
          - <UML:Namespace.ownedElement>
            - <UML:Package name="Localizacao" xmi.id="EAPK_7A95DB61_7BA4_4512_BA18_EDD00253EAE5" isRoot="true"
                isLeaf="false" isAbstract="false" visibility="public">
              - <UML:ModelElement.stereotype>
                  <UML:Stereotype xmi.idref="EAID_E042327D_4778_4c83_B5C2_3BAD986BBD50" />
                </UML:ModelElement.stereotype>
              - <UML:Namespace.ownedElement>
                - <UML:Class name="Pais" xmi.id="EAID_217777D2_8955_435d_AD66_9B010C0D868E" visibility="public"
                    isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false" isActive="false">
                  - <UML:ModelElement.stereotype>
                      <UML:Stereotype xmi.idref="EAID_D8CCE908_DBE2_4a1f_A5A5_5AA2FAC3A630" />
                    </UML:ModelElement.stereotype>
                  - <UML:Classifier.feature>
```


Quadro 4 – Trecho do conteúdo de um artefato XMI

Nota-se que a ferramenta não segue o modelo de pacote fechado, uma das principais características da mesma é a flexibilidade oferecida aos desenvolvedores. Através da utilização da ferramenta uma organização pode adaptar o seu processo de Engenharia da Produção (*Forward Engineering*) conforme as necessidades e particularidades da mesma.

Ao contrário das soluções disponíveis no mercado, a ferramenta não trabalha com a utilização de *scripts* para implementação das rotinas de geração. As rotinas com a lógica para geração de código são construídas através de *plugins*, módulos DLL (*Dynamic Link Library*) compilados através da plataforma Microsoft .NET. Um dos principais fatores para a escolha desta plataforma foi a possibilidade da utilização de diferentes linguagens de programação para construção dos mesmos. Desta forma as organizações que desejarem criar os seus *plugins* podem realizar a implementação dos mesmos utilizando a linguagem de programação mais próxima da sua realidade.

Os principais motivos da utilização de *plugins* compilados ao invés de *scripts* são: o considerável aumento de velocidade na geração e o ganho de recursos para construção dos mesmos, sendo possível desta forma utilizar todos os subsídios fornecidos pela plataforma Microsoft .NET. Dentre eles pode-se citar a orientação a objetos e a utilização de uma linguagem de programação já conceituada no mercado como Java, Visual Basic, Delphi e C#, abstraindo desta forma a necessidade do estudo de uma nova tecnologia ou *script* por parte do desenvolvedor. A necessidade do estudo de novas tecnologias para construção de *scripts* é

perceptível no trabalho correlato de Schmidt (2001), que utiliza uma pseudolinguagem em língua portuguesa para implementação dos mesmos.

Conforme o caso de uso 1.2 (Manter Arquiteturas) a associação do *plugin* de geração com a ferramenta é realizada através da arquitetura utilizada para implementar um determinado software. Ao realizar a chamada para a geração, a ferramenta realiza a carga do *plugin* conforme a arquitetura selecionada, passando ao mesmo toda a estrutura do diagrama de classes importado juntamente com as suas propriedades de especialização, o *plugin* realiza o tratamento das informações recebidas retornando como resultado o código gerado.

O Quadro 5 apresenta a interface de implementação necessária para construção de uma classe de geração de código. Nota-se que a classe que irá conter a lógica de geração de código deverá implementar a interface "IQuelleControlUMLClassModelGeneration". Esta interface é composta pela operação "ExecuteGeneration" que recebe como parâmetro a interface "IQuelleControlUMLClassModel".

Através da interface "IQuelleControlUMLClassModel" se é possível obter através de coleções de dados todos os elementos do modelo de classes, juntamente com os seus atributos e propriedades de especialização. Esta interface implementa a estrutura da classe de controle "ModeloClasses", demonstrada através do diagrama de classes da camada de lógica da ferramenta. Este diagrama pode ser visualizado através da Figura 10 que se encontra no capítulo de especificação da ferramenta.

```
public interface IQuelleControlUMLClassModelGeneration
{
    bool ExecuteGeneration(IQuelleControlUMLClassModel classModel);
}
```

Quadro 5 – Interface para construção de um *plugin*

O Quadro 6 apresenta o exemplo de uma classe básica para geração de código. Nota-se que a mesma implementa a interface apresentada no Quadro 5. O método "ExecuteGeneration" será o responsável por realizar a criação da sintaxe do código gerado, retornando a ferramenta através do seu resultado, se o processo de geração pode ser realizado ou não.

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Text;
using Quelle.Logic.UML;

#endregion

namespace Quelle.Generation
{
    public class CodeGeneration: IQuelleControlUMLClassModelGeneration
    {
        public bool ExecuteGeneration(IQuelleControlUMLClassModel classModel)
        {
            return false;
        }
    }
}
```

Quadro 6 – Exemplo de uma classe básica para construção de um *plugin*

Através da Figura 15 pode-se visualizar a estrutura da pasta de implantação da ferramenta. Por utilizar a tecnologia Microsoft .NET pode-se observar que o tamanho dos arquivos da ferramenta com formato DLL (*Dynamic Link Library*) e EXE (*Executable*) possuem tamanhos consideravelmente reduzidos, quando comparados a arquivos gerados pelas demais tecnologias existentes no mercado. O arquivo com extensão "fdb" é o banco de dados da ferramenta, os arquivos com extensão "dll" os módulos (genérico, persistência, lógica e apresentação) e os arquivos com extensão "xml" se referem respectivamente ao arquivo de configuração da ferramenta e ao arquivo de tradução para o idioma português.

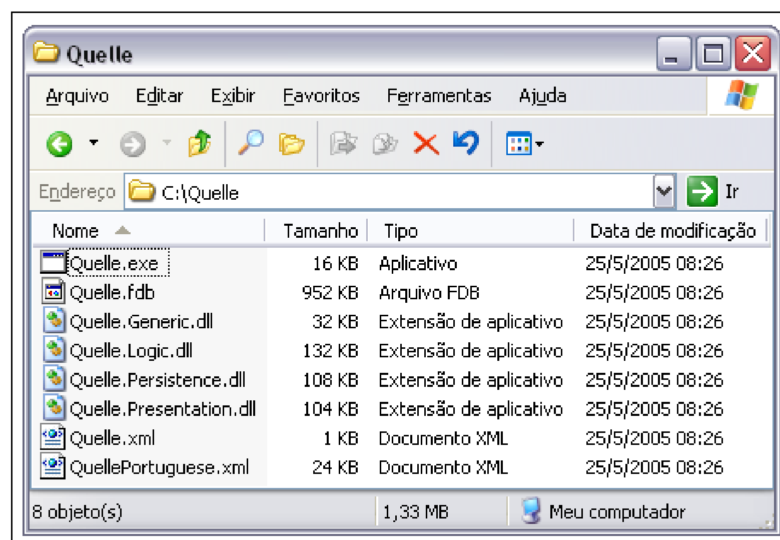


Figura 15 – Estrutura da pasta de implantação da ferramenta

3.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Este tópico apresenta um estudo de caso, do ponto de vista de usuário, objetivando mostrar a funcionalidade e operacionalidade da ferramenta. O conteúdo apresentado é composto por um resumo do caso de uso, pela indicação do seu detalhamento encontrado no Apêndice A e pela apresentação da interface do mesmo.

A Figura 16 apresenta a interface principal da ferramenta e o *menu* de acesso os casos de uso de manutenção. Este *menu* é composto pelas manutenções dos casos de uso considerados como básicos (Projetos, Arquiteturas, Idiomas, Tipos de Dado e Grupos de Propriedades) e pelos casos de uso das manutenções de especialização, apresentados em um segundo *menu*. Maiores detalhes sobre estes casos de uso serão apresentados a seguir.

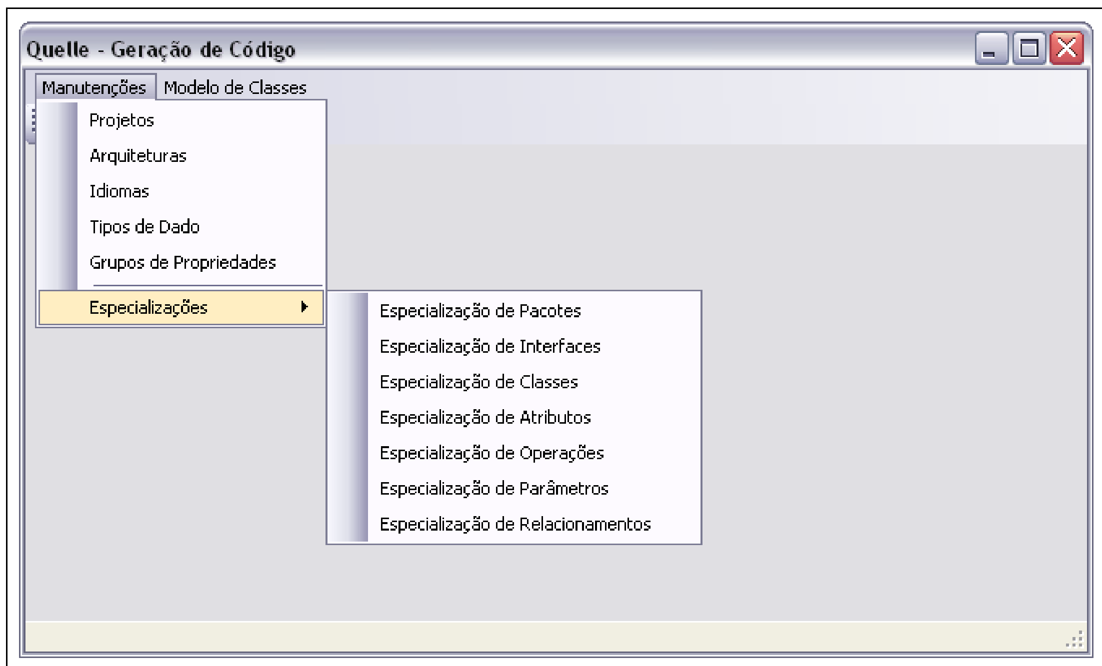


Figura 16 – *Menu* de manutenções

A Figura 17 apresenta a interface principal da ferramenta e o *menu* de acesso aos casos de uso do modelo de classes. Este *menu* é composto pelo caso de uso de especialização do modelo de classes e pelo caso de uso de geração de código. Os casos de uso do modelo de classes são dependentes dos casos de uso considerados como básicos, desta forma os primeiros casos de uso aos quais os usuários devem interagir são os casos de uso considerados como básicos.

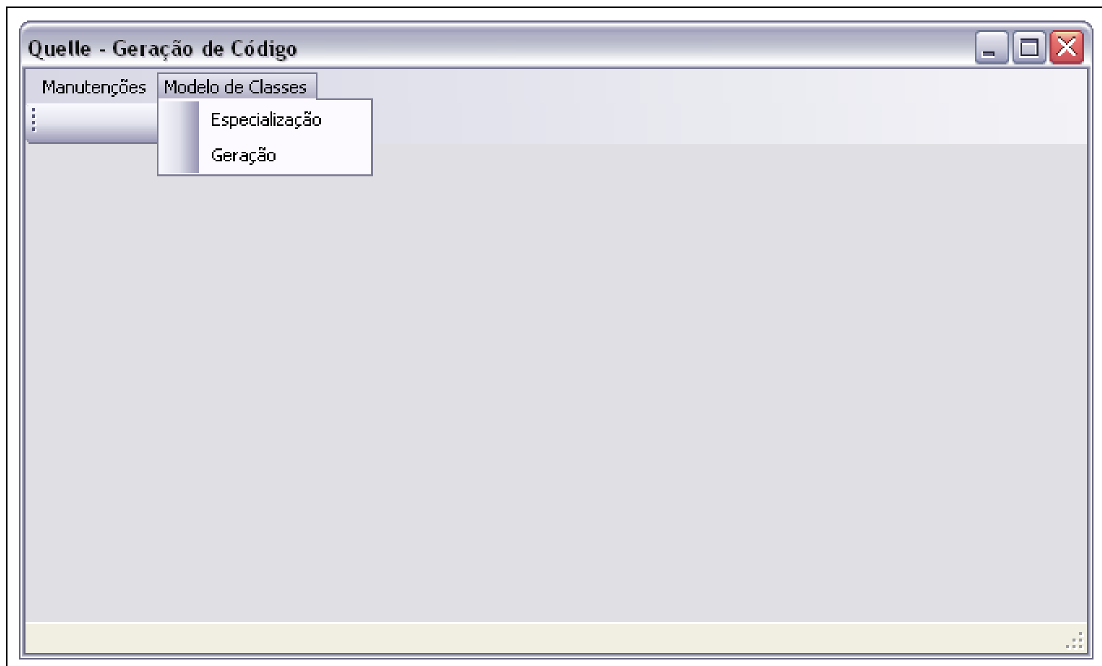


Figura 17 – *Menu* do modelo de classes

A Figura 18 apresenta a interface do caso de uso 1.1 (Manter Projetos). Através deste caso de uso o usuário realiza a manutenção de projetos, cada projeto corresponde a um modelo de classes referenciado através de um artefato XMI (*XML Metadata Interchange*).

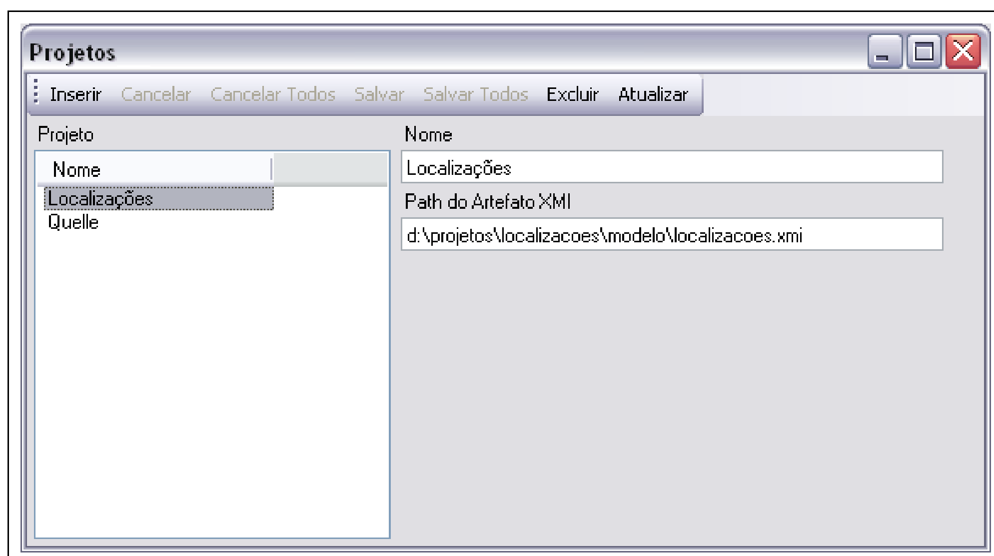


Figura 18 – Manutenção de projetos

A Figura 19 apresenta a interface do caso de uso 1.2 (Manter Arquiteturas). Através deste caso de uso o usuário realiza a manutenção de arquiteturas utilizando o conceito de herança. Cada arquitetura possui o *path* para o seu respectivo *plugin* de geração.

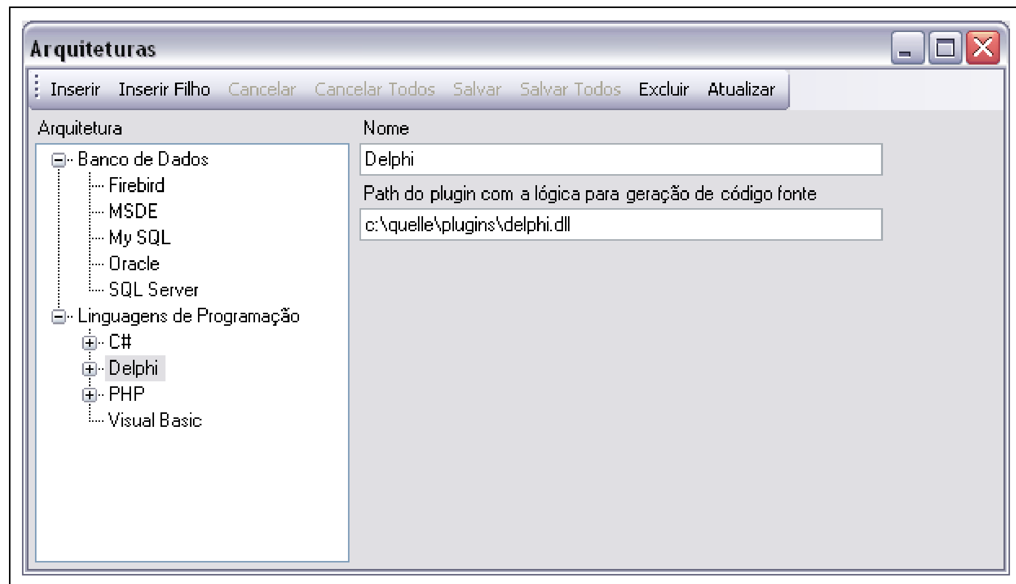


Figura 19 – Manutenção de arquiteturas

A Figura 20 apresenta a interface do caso de uso 1.3 (Manter Idiomas). Através deste caso de uso o usuário realiza a manutenção de idiomas utilizando o conceito de herança.

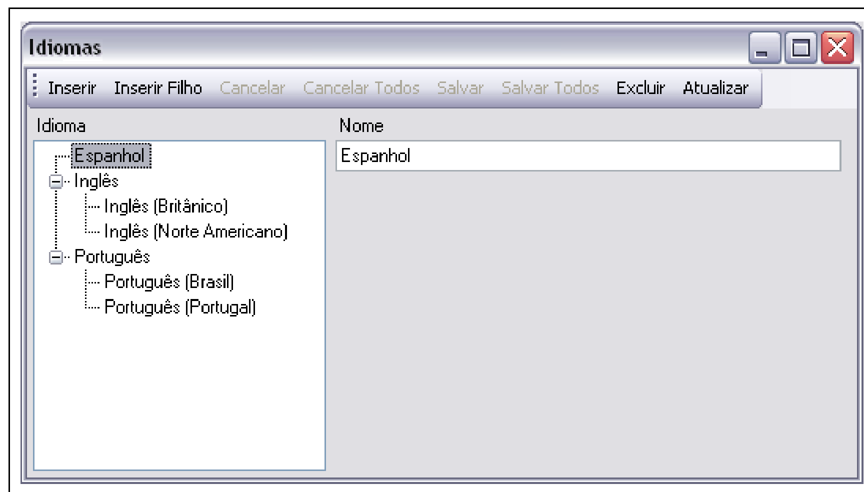


Figura 20 – Manutenção de idiomas

A Figura 21 apresenta a interface do caso de uso 1.4 (Manter Tipos de Dado). Através deste caso de uso o usuário realiza a manutenção de tipos de dado utilizando o conceito de herança. O caso de uso 1.5 (Manter Propriedades de Tipo de Dado) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de um tipo de dado na árvore de navegação.

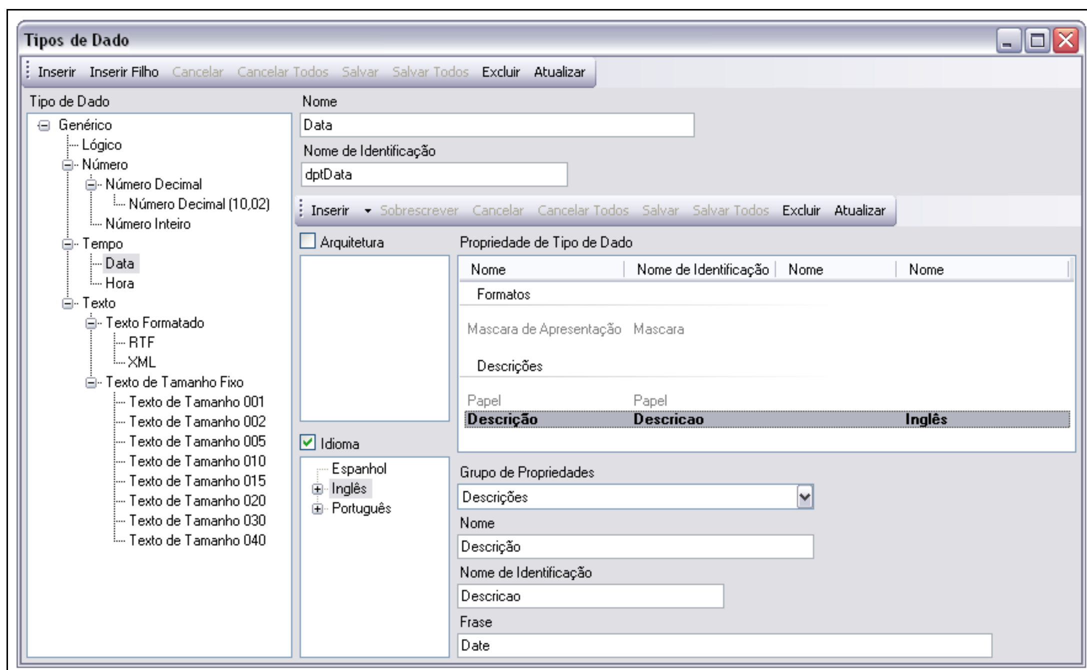


Figura 21 – Manutenção de tipos de dado

A Figura 22 apresenta a interface do caso de uso 1.6 (Manter Grupos de Propriedades). Através deste caso de uso o usuário realiza a manutenção de grupos de propriedades. Os grupos de propriedades são utilizados para agrupar as propriedades de especialização. Um exemplo é a utilização do grupo de propriedades “Tamanhos” para o agrupamento das propriedades “Altura” e “Largura”.

Figura 22 – Manutenção de grupos de propriedades

A Figura 23 apresenta a interface do caso de uso 02.01.01 (Manter Especialização de Pacotes). Através deste caso de uso o usuário realiza a manutenção das especializações de

pacotes utilizando o conceito de herança. O caso de uso 02.02.01 (Manter Propriedades de Especialização de Pacotes) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de pacote na árvore de navegação.

No exemplo apresentado na Figura 23 a especialização de pacote "Módulo" possui várias propriedades herdadas do ancestral "Genérico", mas para a arquitetura "Delphi" foi criada uma nova propriedade chamada "Extensão do Pacote" e para a mesma foi atribuído o valor "BPL".

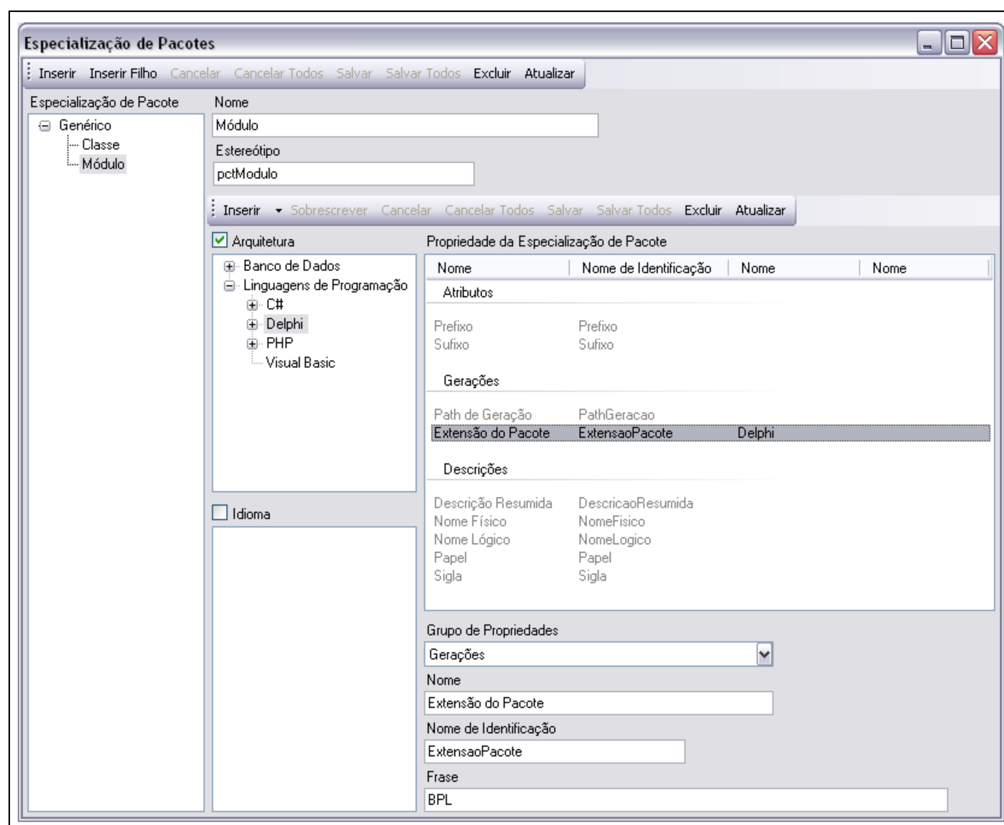


Figura 23 – Manutenção da especialização de pacotes

A Figura 24 apresenta a interface do caso de uso 02.01.02 (Manter Especialização de Interfaces). Através deste caso de uso o usuário realiza a manutenção das especializações de interfaces utilizando o conceito de herança. O caso de uso 02.02.02 (Manter Propriedades de Especialização de Interfaces) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de interface na árvore de navegação.

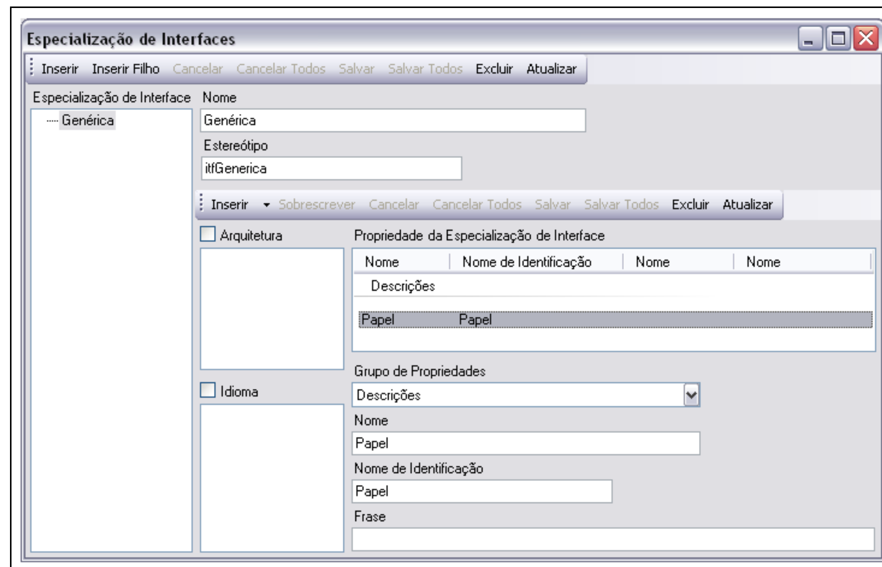


Figura 24 – Manutenção da especialização de interfaces

A Figura 25 apresenta a interface do caso de uso 02.01.03 (Manter Especialização de Classes). Através deste caso de uso o usuário realiza a manutenção das especializações de classes utilizando o conceito de herança. O caso de uso 02.02.03 (Manter Propriedades de Especialização de Classes) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de classe na árvore de navegação.

Nota-se que a especialização selecionada “Interface Gráfica” é descendente da especialização “Fronteira”, como exemplo, a especialização “Interface Gráfica” poderia simbolizar um Formulário quando aplicado a linguagem de programação C# ou uma página de navegação internet, quando aplicada a linguagem de programação PHP (*Hypertext Preprocessor*).

Levando em consideração que a especialização “Interface Gráfica” possui uma propriedade de especialização “Altura”, no exemplo indicado na Figura 25 a mesma está sendo sobrescrita para linguagem de programação C# com o valor “80”, desta forma todas as especializações descendentes de “Interface Gráfica” possuirão a propriedade herdada “Altura” com valor igual a “80”.

Seguindo o exemplo proposto, ao realizar a importação de um modelo de classes através do caso de uso 3.1 (Importar Modelo de Classes), toda classe com estereótipo igual a “clsInterfaceGrafica” receberá como ancestral a especialização “Interface Gráfica”, herdando desta forma todas as suas propriedades de especialização, inclusive a propriedade “Altura” definida para a linguagem de programação C# com o valor igual a “80”.

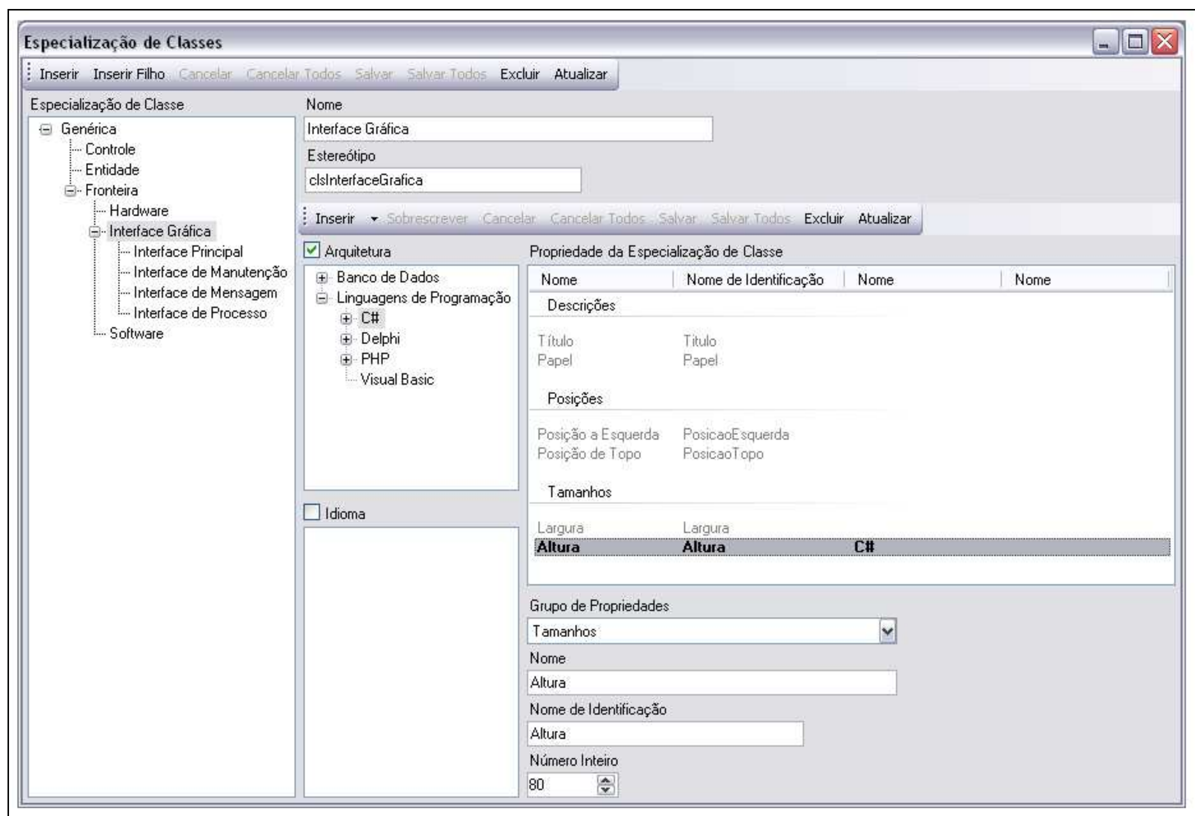


Figura 25 – Manutenção da especialização de classes

A Figura 26 apresenta a interface do caso de uso 02.01.04 (Manter Especialização de Atributos). Através deste caso de uso o usuário realiza a manutenção das especializações de atributos utilizando o conceito de herança. O caso de uso 02.02.04 (Manter Propriedades de Especialização de Atributos) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de atributo na árvore de navegação.

No exemplo apresentado na Figura 26 as propriedades da especialização do atributo "Nome" estão sendo sobrescritas para o idioma português. A identificação de uma propriedade sobrescrita pode ser identificada através da utilização da característica "negrito", aplicada a propriedade de especialização.

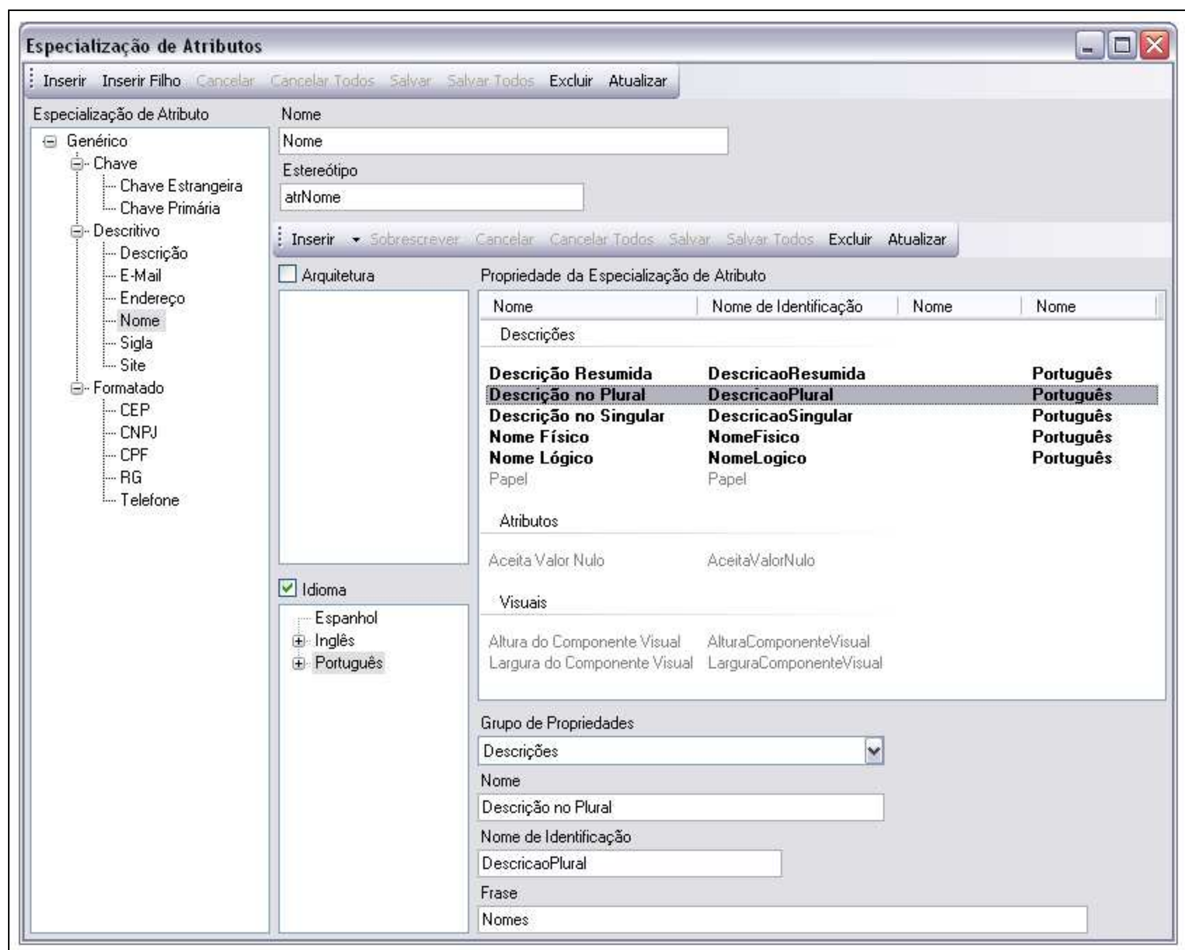


Figura 26 – Manutenção da especialização de atributos

A Figura 27 apresenta a interface do caso de uso 02.01.05 (Manter Especialização de Operações). Através deste caso de uso o usuário realiza a manutenção das especializações de operações utilizando o conceito de herança. O caso de uso 02.02.05 (Manter Propriedades de Especialização de Operações) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de operação na árvore de navegação.

O exemplo apresentado na Figura 27 demonstra a propriedade "Esboço da Implementação", sendo que o usuário poderia realizar nesta propriedade comentários de como a operação deve ser implementada, podendo esta propriedade servir tanto para a documentação do software como para a consulta do programador encarregado pela implementação da mesma.

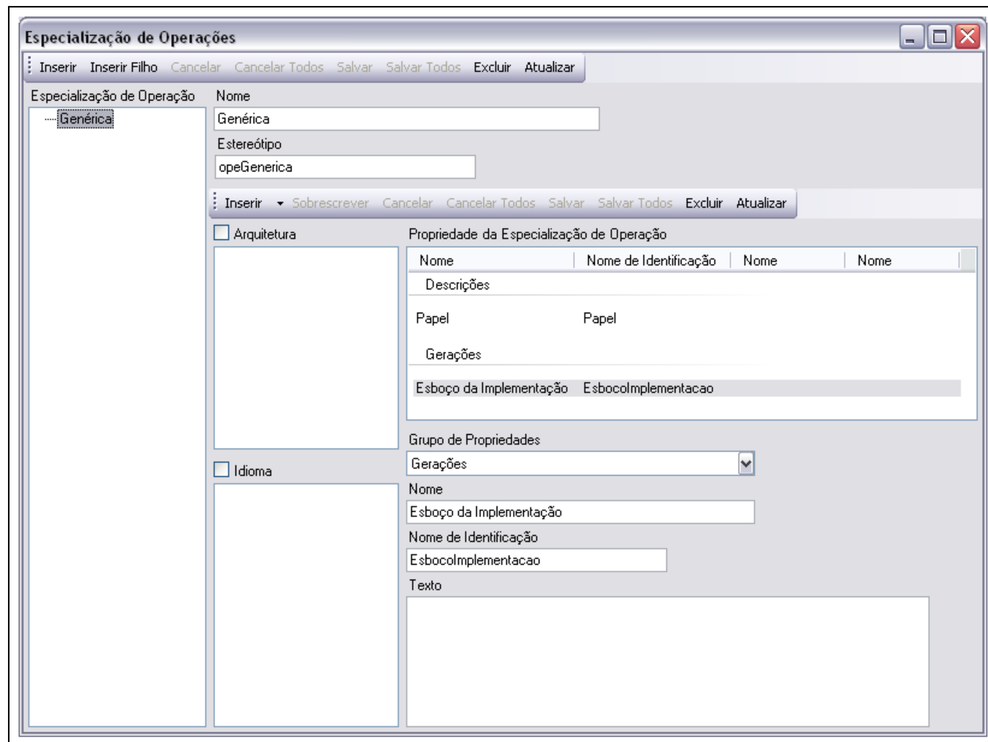


Figura 27 – Manutenção da especialização de operações

A Figura 28 apresenta a interface do caso de uso 02.01.06 (Manter Especialização de Parâmetros). Através deste caso de uso o usuário realiza a manutenção das especializações de parâmetros utilizando o conceito de herança. O caso de uso 02.02.06 (Manter Propriedades de Especialização de Parâmetros) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de parâmetros na árvore de navegação.

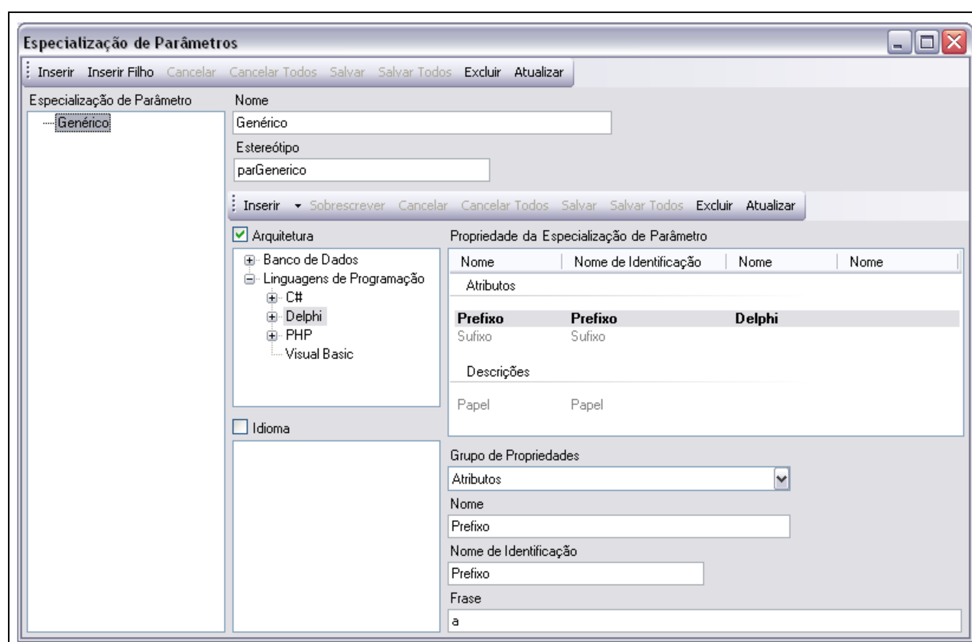


Figura 28 – Manutenção da especialização de parâmetros

A Figura 29 apresenta a interface do caso de uso 02.01.07 (Manter Especialização de Relacionamentos). Através deste caso de uso o usuário realiza a manutenção das especializações de relacionamentos utilizando o conceito de herança. O caso de uso 02.02.07 (Manter Propriedades de Especialização de Relacionamentos) também é contemplado por esta interface. Este caso de uso é ativado através da escolha de uma especialização de relacionamento na árvore de navegação.

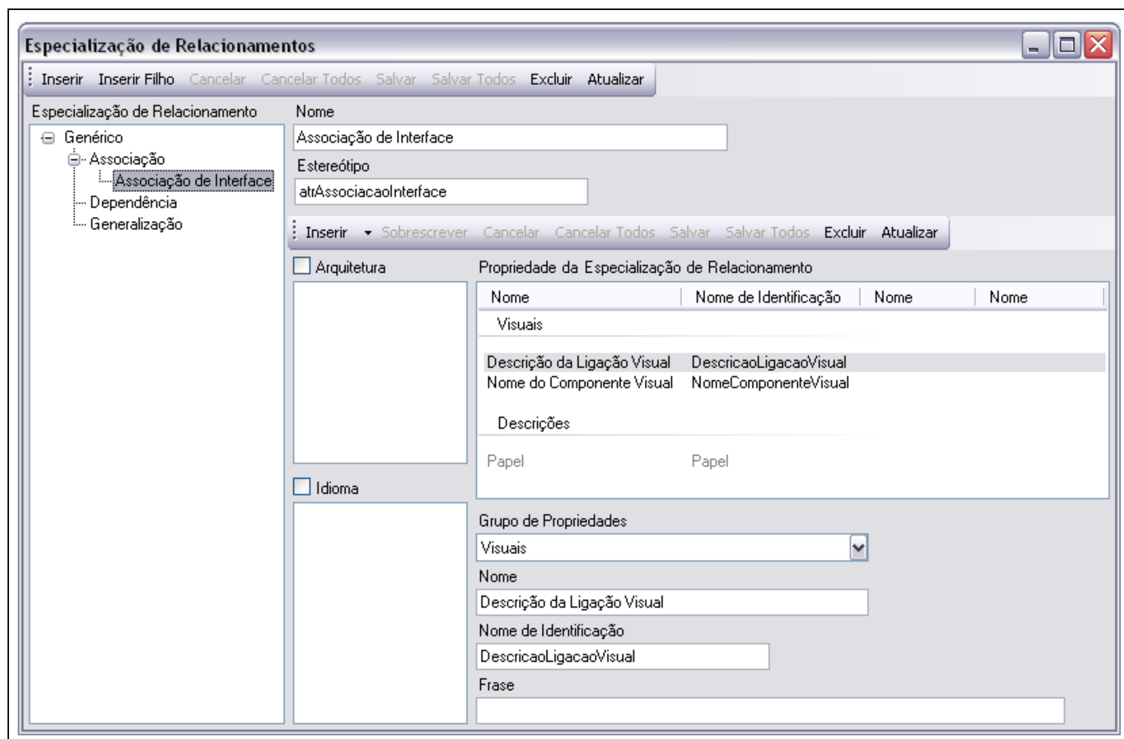


Figura 29 – Manutenção da especialização de relacionamentos

A Figura 30 apresenta a interface do caso de uso 3.2 (Especializar Modelos de Classes). Através deste caso de uso o usuário realiza a especialização dos elementos do modelo de classes (Pacotes, Interfaces, Atributos, Operações, Parâmetros e Relacionamentos). O usuário escolhe o projeto a ser especializado e a ferramenta realiza a importação do mesmo através de um artefato XMI indicado pelo projeto. O modelo de classes é apresentado para o usuário através de uma árvore de navegação, ao clicar sobre um dos elementos do modelo de classes a ferramenta apresenta os seus atributos e as propriedades de especialização.

O exemplo apresentado na Figura 30 demonstra a especialização de um modelo de classes composto por um pacote denominado “Persistência”. Este pacote contém três classes (“Pais”, “Estado” e “Municipio”). No exemplo, a propriedade de especialização “Descrição no Plural” do atributo “nome” está sendo sobrescrito para o idioma português.

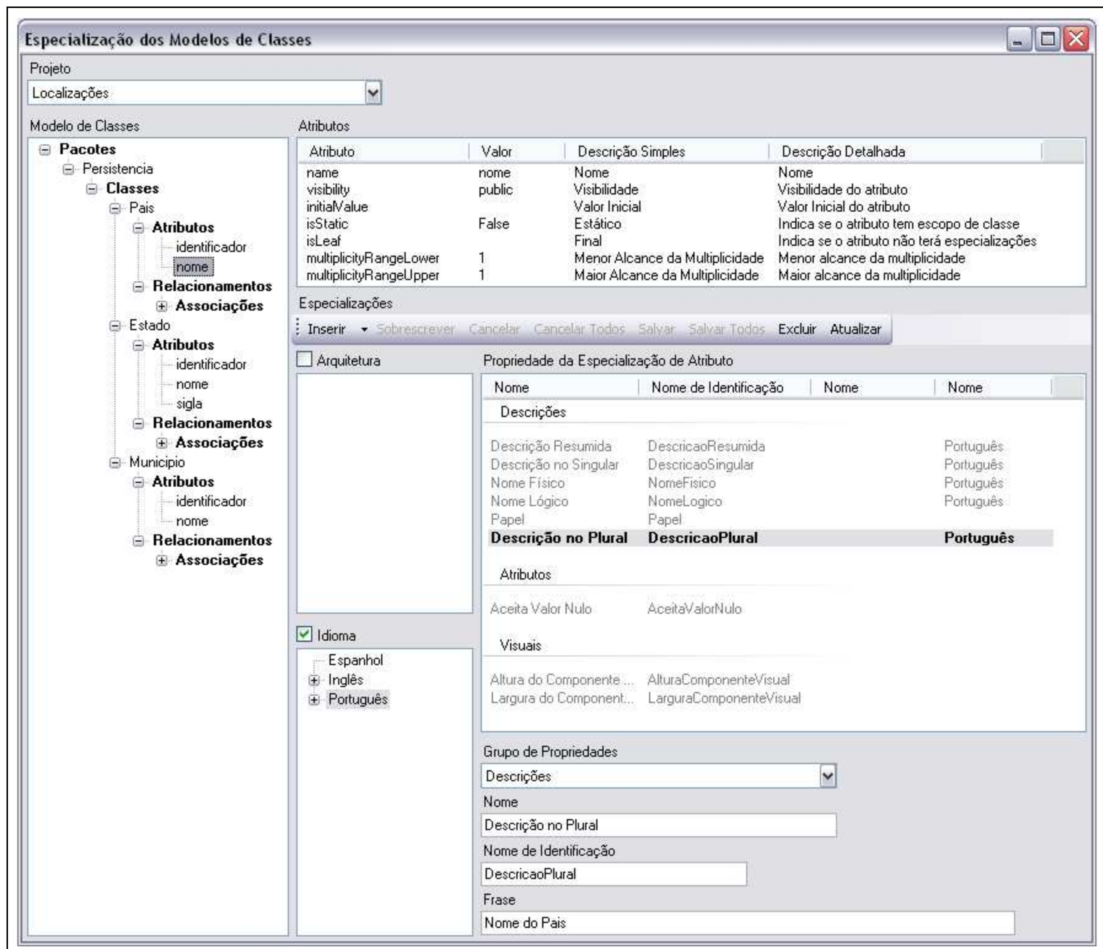


Figura 30 – Especialização dos modelos de classes

A Figura 31 apresenta a interface do caso de uso 3.3 (Geração de Código). Através deste caso de uso o usuário realiza a escolha do projeto, pacotes e das classes a serem geradas, escolhendo também a arquitetura e o idioma da geração.

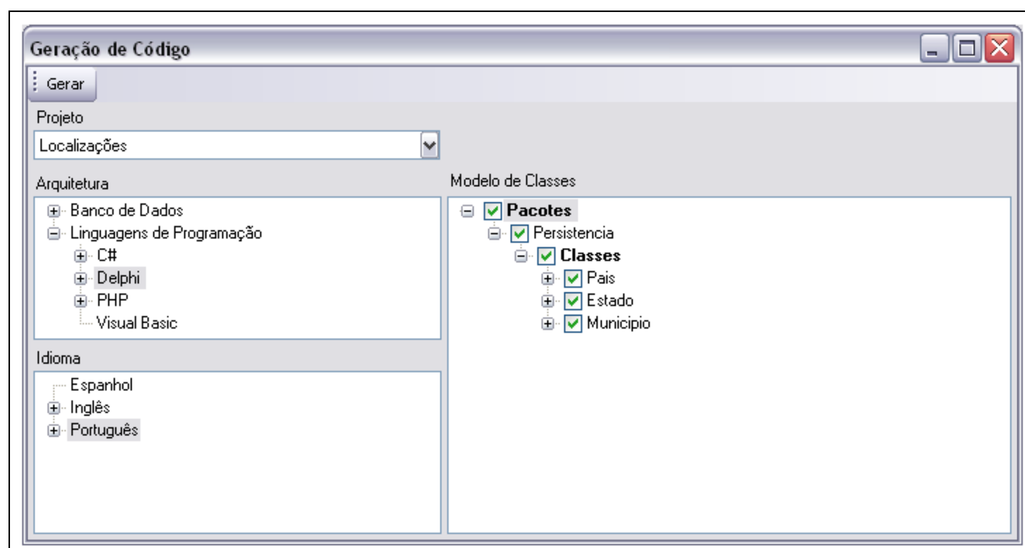


Figura 31 – Geração de código

3.3.3.1 Geração de código e *plugins*

Este tópico tem o objetivo de apresentar um exemplo de geração de código com base em um diagrama de classes de localizações. Este exemplo demonstra a capacidade de especialização da ferramenta por arquitetura e idioma. Para a especialização por arquitetura foram utilizadas as arquiteturas Java e PCL (*PHP Custom Library*), uma biblioteca para desenvolvimento em PHP (*Hypertext Preprocessor*) fornecida pela Ápice Engenharia de Software. Na especialização por idioma foram utilizados os idiomas português e inglês. Outro objetivo é demonstrar a diferença entre um código gerado no padrão das ferramentas CASE existentes no mercado e um código gerado através da ferramenta proposta pelo trabalho. A Figura 32 apresenta o diagrama de classes de localizações, este diagrama possui três classes de entidade (“Pais”, “Estado” e “Municipio”).

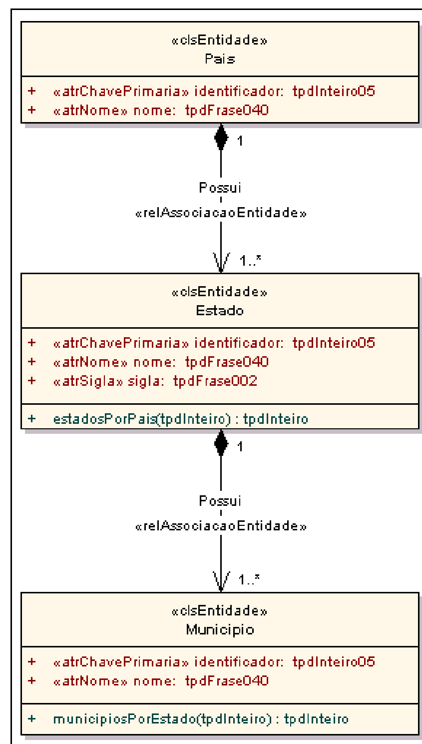


Figura 32 – Diagrama de localizações

Para a apresentação deste exemplo foram construídos dois *plugins* para geração de código, um para a arquitetura Java denominado “QuelleGenerationJava” e outro para a arquitetura PCL denominado “QuelleGenerationPCL”. A geração de código para a arquitetura Java tem o objetivo de gerar um código semelhante ao código gerado pelas ferramentas CASE (*Computer Aided Software Engineering*) existentes no mercado. O Quadro 7 apresenta um trecho do código escrito para o *plugin* de geração “QuelleGenerationJava”.

```
for (i = 0; i < elementClass.Attributes.Count; i++) 1
{
    elementAttribute = elementClass.Attributes.Item(i);

    if (elementAttribute.Enabled)
    {
        attributeName = elementAttribute.ElementAttributes.Name.Value.ToString();
        dataType = elementAttribute.DataType.ElementSpecializations.DataStringOfIdentifyName(JAVA_DATA_TYPE).Value; 2

        javaVariant = javaClass.Variants.Add(attributeName,
                                     dataType,
                                     JavaFontConstants.JAVA_DATA_VALUE_NULL,
                                     javaClass.Variants.Visibilities.PrivateVisibility);

        // Implementa método "GET" para o atributo
        methodName = (JavaFontConstants.JAVA_PREFIX_GET +
                     this.TransformInitialTextUpper(attributeName));

        javaMethod = javaClass.Methods.Add(methodName,
                                   dataType,
                                   this.ElementVisibilityToJavaVisibility(javaFont.Classes, elementAttribute),
                                   this.ElementModifiersToJavaModifiers(javaFont.Classes, elementAttribute));

        javaMethod.Implementation.Add(JavaFontConstants.JAVA_FONT_COMMAND_RETURN +
                              CustomConstants.STRING_SPACE +
                              JavaFontConstants.JAVA_VARIABLE_THIS +
                              JavaFontConstants.JAVA_OPERATOR_CLASS_DELIMITER +
                              javaVariant.Name +
                              JavaFontConstants.JAVA_FONT_COMMAND_SEPARATOR);

        // Implementa método "SET" para o atributo

        methodName = (JavaFontConstants.JAVA_PREFIX_SET +
                     this.TransformInitialTextUpper(attributeName));

        javaMethod = javaClass.Methods.Add(methodName,
                                   JavaFontConstants.JAVA_DATA_TYPE_VOID,
                                   javaClass.Methods.Visibilities.PublicVisibility,
                                   null);

        javaParameter = javaMethod.Parameters.Add(attributeName, dataType);

        javaMethod.Implementation.Add(JavaFontConstants.JAVA_VARIABLE_THIS +
                              JavaFontConstants.JAVA_OPERATOR_CLASS_DELIMITER +
                              javaVariant.Name +
                              CustomConstants.STRING_SPACE +
                              JavaFontConstants.JAVA_OPERATOR_ATTRIBUTION +
                              CustomConstants.STRING_SPACE +
                              javaParameter.Name +
                              JavaFontConstants.JAVA_FONT_COMMAND_SEPARATOR);
    }
}
```

Quadro 7 – Trecho do código escrito para o *plugin* de geração da arquitetura Java

Nota-se que o destaque “1” do trecho de código apresentado no Quadro 7 demonstra um *loop* que percorre de todos os atributos de uma determinada classe visando à implementação do método de retorno “*get*” e do método de atribuição “*set*” para cada atributo percorrido. Nota-se também através do destaque “2” a busca pelo valor da propriedade de especialização “Tipo de Dado”.

Como exemplo para o destaque “2” citado anteriormente, a Figura 33 apresenta a propriedade de especialização “Tipo de Dado” para a especialização do tipo de dado “Inteiro”. Nota-se que este tipo de dado é utilizado pelo diagrama de localizações apresentado na Figura 32. Nota-se também que a propriedade em questão está sendo sobrescrita para a arquitetura Java com o valor “int”.

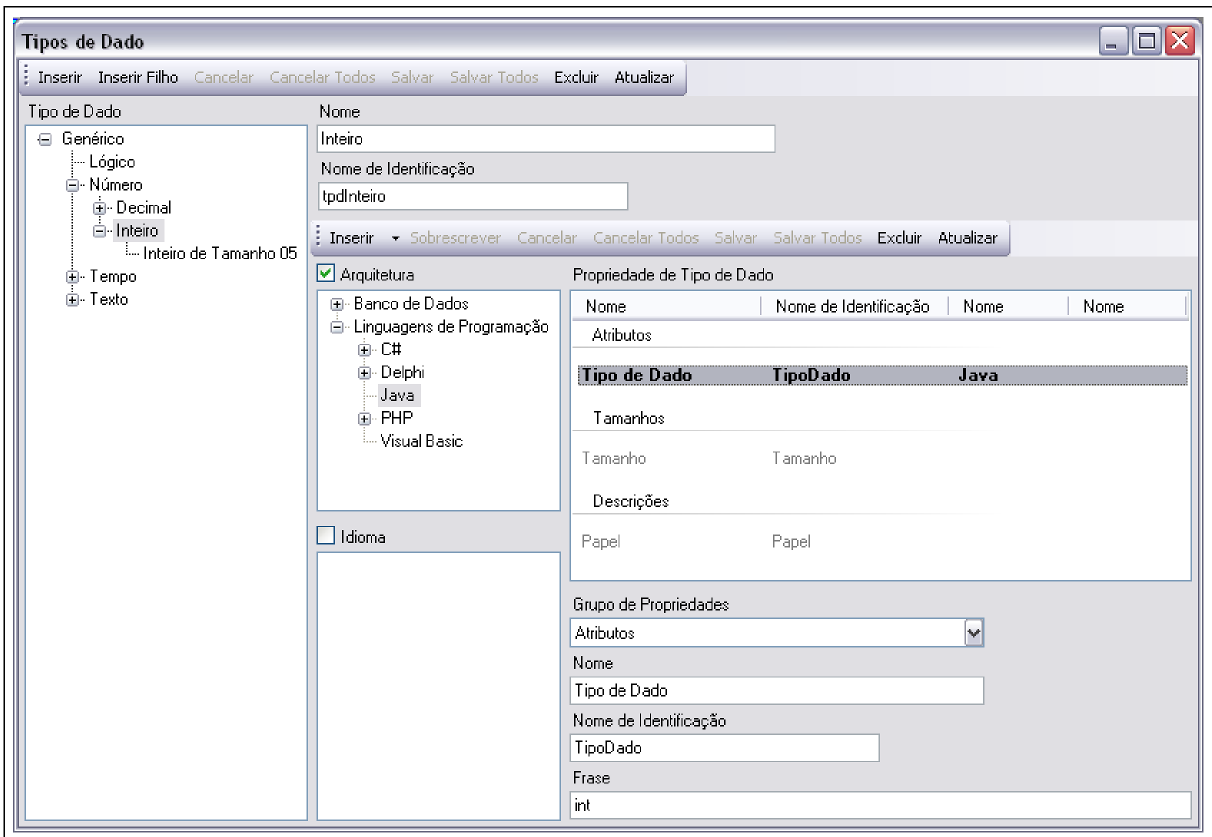


Figura 33 – Exemplo da especialização do tipo de dado “Inteiro”

O Quadro 8 apresenta um trecho do código gerado para a classe “Estado”. Nota-se em destaque a utilização do tipo de dado “Inteiro” anteriormente citado. O exemplo completo da geração de código para a arquitetura Java encontra-se no Anexo A. Encontra-se também no Anexo A um exemplo do código gerado através da ferramenta Enterprise Architect. Este exemplo comprova a semelhança entre os dois códigos.

```
/*
Fonte Gerado Automaticamente
Plugin: QuelleGenerationJava
Data: 6/6/2005
Hora: 18:16
*/

public class Estado {

    private int identificador = null;
    private String nome = null;
    private String sigla = null;
    private Pais pais = null;

    public Estado() {
        super();
    }

    public int getIdentificador() {
        return this.identificador;
    }
```

Quadro 8 – Trecho de código Java gerado para a classe “Estado”

A geração de código para a arquitetura PCL tem o objetivo de demonstrar uma geração de código mais completa e refinada. O Quadro 9 apresenta um trecho do código escrito para o *plugin* de geração “QuelleGenerationPCL”.

```
private bool GenerateClassEntity(IQuelleControlUMLClassModelClass elementClass, string path)
{
    try
    {
        IPHPFont phpFont = new PHPFont();
        IQuelleControlUMLClassModelAttribute elementAttribute = null;
        IQuelleControlUMLClassModelRelationAssociation[] associations = null;
        IQuelleControlUMLClassModelRelationAssociation elementAssociation = null;
        IPHPClass phpClassEntity = null;
        IPHPClass phpClassAttributes = null;
        IPHPClass phpClassViews = null;
        IPHPClass phpClassRelations = null;
        IPHPClass phpClassViewLookup = null;
        IPHPClass phpClassViewList = null;
        IPHPClass phpClassViewMaintenance = null;
        IPHPMethod phpMethodEntity = null;
        IPHPMethod phpMethodAttributes = null;
        IPHPMethod phpMethodViews = null;
        IPHPMethod phpMethodRelations = null;
        IPHPMethod phpMethodViewLookup = null;
        IPHPMethod phpMethodViewList = null;
        IPHPMethod phpMethodViewMaintenance = null;
        string classEntityName = elementClass.ElementAttributes.Name.Value.ToString();
        string classAttributesName = (classEntityName + PCL_CLASS_ATTRIBUTES_SUFIX);
        string classViewsName = (classEntityName + PCL_CLASS_VIEWS_SUFIX);
        string classRelationsName = (classEntityName + PCL_CLASS_RELATIONS_SUFIX);
        string classViewLookupName = (classEntityName + PCL_VIEW_SUFIX + PCL_VIEW_LOOKUP_SUFIX);
        string classViewListName = (classEntityName + PCL_VIEW_SUFIX + PCL_VIEW_LIST_SUFIX);
        string classViewMaintenanceName = (classEntityName + PCL_VIEW_SUFIX + PCL_VIEW_MAINTENANCE_SUFIX);
        string text = CustomConstants.STRING_EMPTY;
        int i = 0;

        // Arquivos requeridos

        phpFont.References.Add(PHPFontConstants.ClassToFileName(PCL_CUSTOM_ENTITY));
        phpFont.References.Add(PHPFontConstants.ClassToFileName(PCL_CUSTOM_ENTITY_ATTRIBUTES));
        phpFont.References.Add(PHPFontConstants.ClassToFileName(PCL_CUSTOM_ENTITY_VIEWS));
        phpFont.References.Add(PHPFontConstants.ClassToFileName(PCL_CUSTOM_ENTITY_RELATIONS));

        // Cria a classe de atributos

        phpClassAttributes = this.AddClassDefault(null, phpFont, classAttributesName, PCL_CUSTOM_ENTITY_ATTRIBUTES);
        phpMethodAttributes = phpClassAttributes.Methods.Item(0);

        // Cria as classes de visões padrões

        phpClassViewLookup = this.AddClassDefault(null, phpFont, classViewLookupName, PCL_CUSTOM_ENTITY_VIEW);
        phpMethodViewLookup = phpClassViewLookup.Methods.Item(0);
        phpMethodViewLookup.Implementation.AddBlank();

        phpClassViewList = this.AddClassDefault(null, phpFont, classViewListName, PCL_CUSTOM_ENTITY_VIEW);
        phpMethodViewList = phpClassViewList.Methods.Item(0);
        phpMethodViewList.Implementation.AddBlank();

        phpClassViewMaintenance = this.AddClassDefault(null, phpFont, classViewMaintenanceName, PCL_CUSTOM_ENTITY_VIEW);
        phpMethodViewMaintenance = phpClassViewMaintenance.Methods.Item(0);
        phpMethodViewMaintenance.Implementation.AddBlank();
```

Quadro 9 – Trecho do código escrito para o *plugin* de geração da arquitetura PCL

O principal motivo para a escolha desta arquitetura é a necessidade da criação de três subclasses (“Atributos”, “Visoes” e “Relacionamentos”) que auxiliam a classe de entidade em questão na manipulação de seus dados.

A subclasse “Atributos” tem o objetivo de fornecer uma lista com todos os atributos para a classe de entidade em questão. Os atributos são adicionados na lista conforme o seu

tipo de dado. O Quadro 10 apresenta um trecho do código gerado para a subclasse “Atributos” da classe “Estado”.

```
class EstadoAtributos extends CustomEntityAttributes {

    var $identificador = NULL;
    var $nome = NULL;
    var $sigla = NULL;
    var $pais = NULL;

    function EstadoAtributos($owner) {
        $this->CustomEntityAttributes($owner);

        $this->identificador =
        $this->addAttributeInteger("identificadorEstado",
                                   "id_est",
                                   "Identificador do Estado",
                                   true,
                                   false,
                                   false,
                                   5,
                                   NULL,
                                   NULL);

        $this->nome =
        $this->addAttributeString("nomeEstado",
                                  "nm_est",
                                  "Nome do Estado",
                                  false,
                                  false,
                                  false,
                                  40,
                                  NULL,
                                  NULL);
```

Quadro 10 – Trecho do código PCL gerado para a subclasse “Atributos” da classe “Estado”

A subclasse “Visoes” tem o objetivo de fornecer uma lista com três visões padrões requeridas pelas classes de entidade da arquitetura PCL. Na estrutura utilizada pela arquitetura PCL uma visão tem o objetivo de fornecer diferentes formas de visualização dos dados armazenados por uma classe de entidade. Cada uma das três visões resulta na necessidade da criação de uma nova subclasse. O Quadro 11 apresenta um trecho do código gerado para a subclasse “Visoes” da classe “Estado”.

```
class EstadoVisoes extends CustomEntityViews {

    var $lookup = NULL;
    var $lista = NULL;
    var $manutencao = NULL;

    function EstadoVisoes($owner) {

        $item = NULL;

        $this->CustomEntityViews($owner);

        $item = new EstadoVisaoLookup($this);
        $this->lookup = $this->add($item, "lookup", "lookup", "lookup");

        $item = new EstadoVisaoLista($this);
        $this->lista = $this->add($item, "lista", "lista", "lista");

        $item = new EstadoVisaoManutencao($this);
        $this->manutencao = $this->add($item, "manutencao", "manutencao", "manutencao");
    }

    function getLookup() {
        return $this->getItem($this->lookup);
    }
```

Quadro 11 – Trecho do código PCL gerado para a subclasse “Visoes” da classe “Estado”

A subclasse “Relacionamentos” tem o objetivo de fornecer a classe de entidade em questão uma lista com as instâncias de todas as entidades com que a mesma se relaciona. Estes relacionamentos auxiliam as classes de entidades na busca de dados em classes com que as mesmas se relacionam. O Quadro 12 apresenta o código gerado para a subclasse “Relacionamentos” da classe “Estado”.

```
class EstadoRelacionamentos extends CustomEntityRelations {

    var $pais = NULL;

    function EstadoRelacionamentos($owner) {

        $item = NULL;

        $this->CustomEntityRelations($owner);

        $item = new Pais($this);
        $this->pais = $this->add($item,
                                 $item->getAttributes()->getIdentificador(),
                                 $item->getViews()->getLookup(),
                                 $this->getEntity()->getAttributes()->getPais());
    }

    function getPais() {
        return $this->getItem($this->pais);
    }

}
```

Quadro 12 – Código PCL gerado para a subclasse “Relacionamentos” da classe “Estado”

As classes de entidade da arquitetura PCL também requerem para a geração de código algumas propriedades como “Nome Lógico”, “Nome Físico” e “Descrição”, que por serem de caráter particular, não são suportadas pelos diagramas de classes UML (*Unified Modeling Language*). Neste caso surge a necessidade da criação das respectivas propriedades na especialização das classes de entidade na ferramenta conforme apresentado na Figura 34.

A Figura 35 apresenta a interface de especialização dos modelos de classes. Nota-se que a mesma demonstra a especialização da classe “Estado”, por ter o mesmo estereótipo que a especialização das classes de entidades demonstrada na Figura 34. A classe “Estado” herda todas as propriedades previamente cadastradas para a mesma.

Também é demonstrado através da Figura 35 a atribuição de um novo valor para a propriedade “Descrição”. Este novo valor é atribuído ao sobrescrever o valor anteriormente cadastrado na especialização das classes de entidade, conforme demonstrado na Figura 34. Nota-se também que o mesmo está sendo especificado para o idioma português.

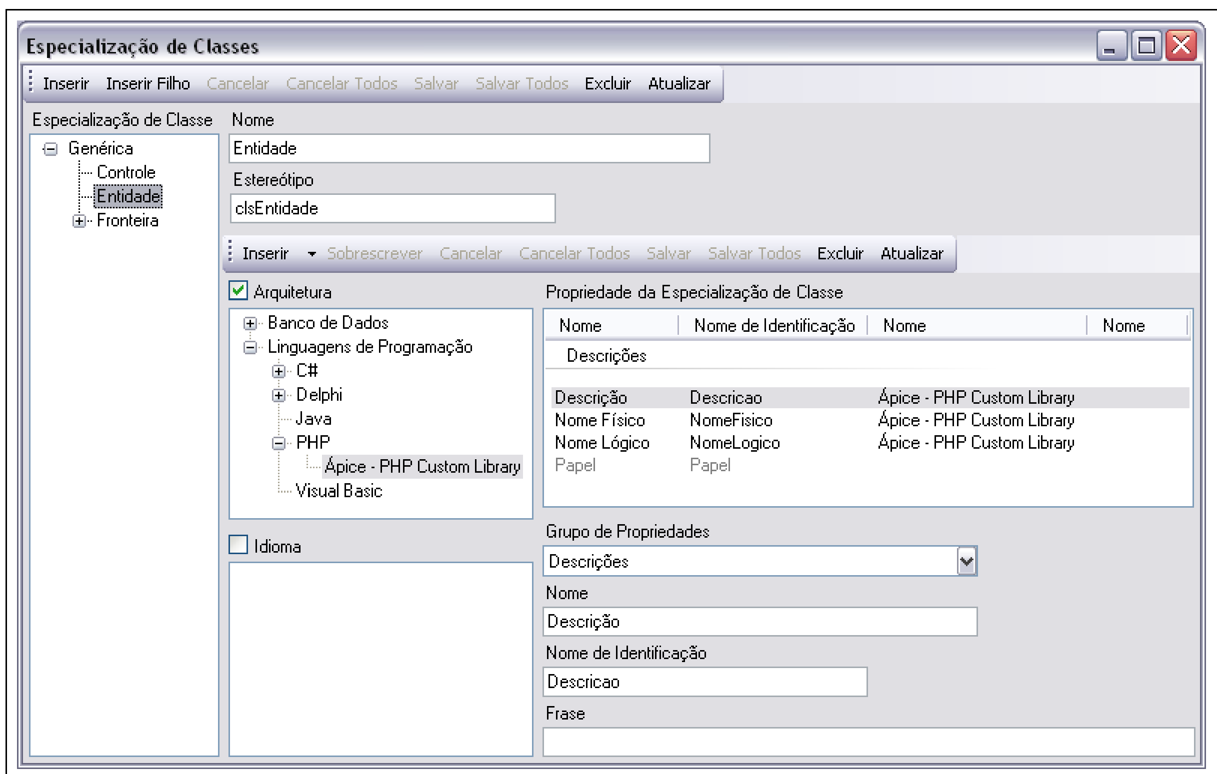


Figura 34 – Propriedades de especialização das classes de entidade

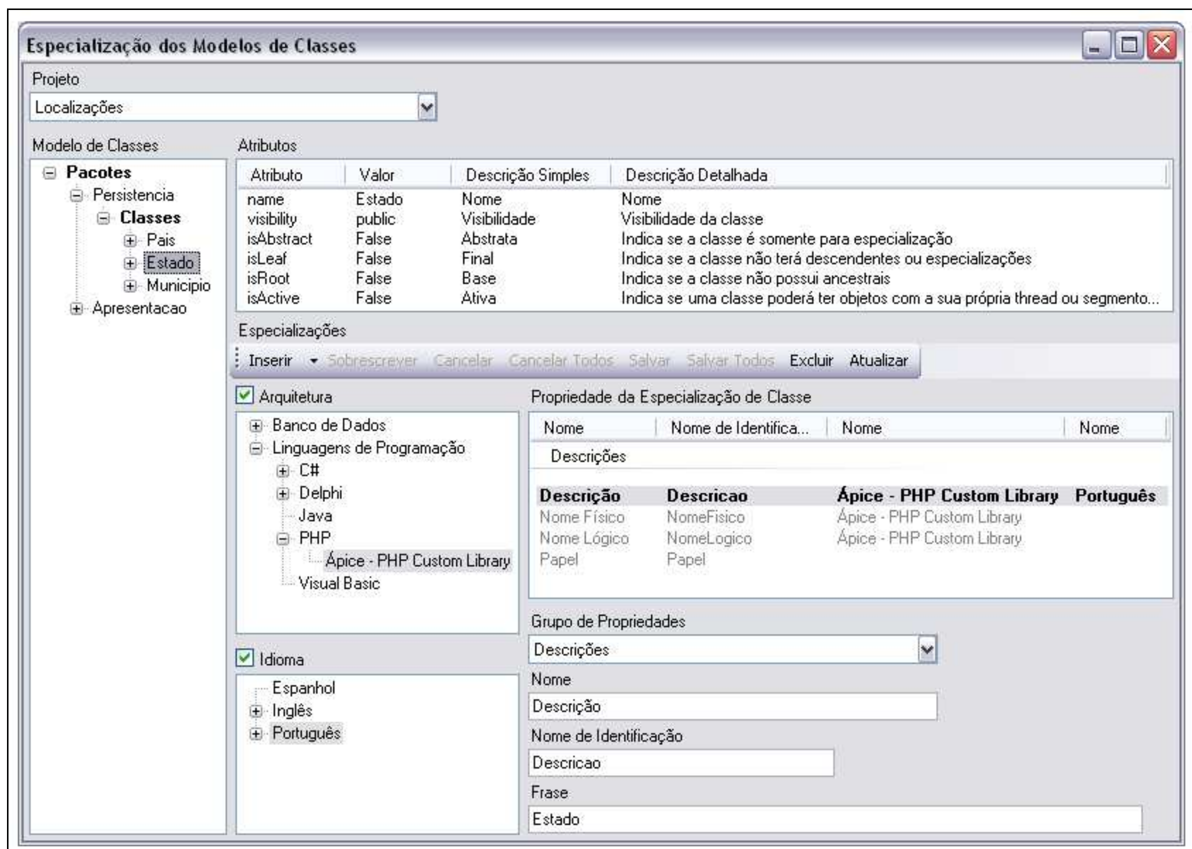


Figura 35 – Especialização da classe “Estado”

A sofisticação tecnológica crescente em diversos países e o intercâmbio mundial de tecnologias, resultante tanto do crescimento tecnológico quanto do processo de globalização da economia, implicam em uma necessidade maior de observação de aspectos internacionais no projeto e desenvolvimento de software (ARAÚJO, 2003, p. 45). Sendo assim a especialização por idioma é um recurso indispensável para organizações que realizam a internacionalização de seus softwares.

A Figura 36 apresenta a especialização da propriedade "Descrição" para o idioma inglês. Nota-se que na Figura 35 a mesma propriedade está especializada para o idioma português.

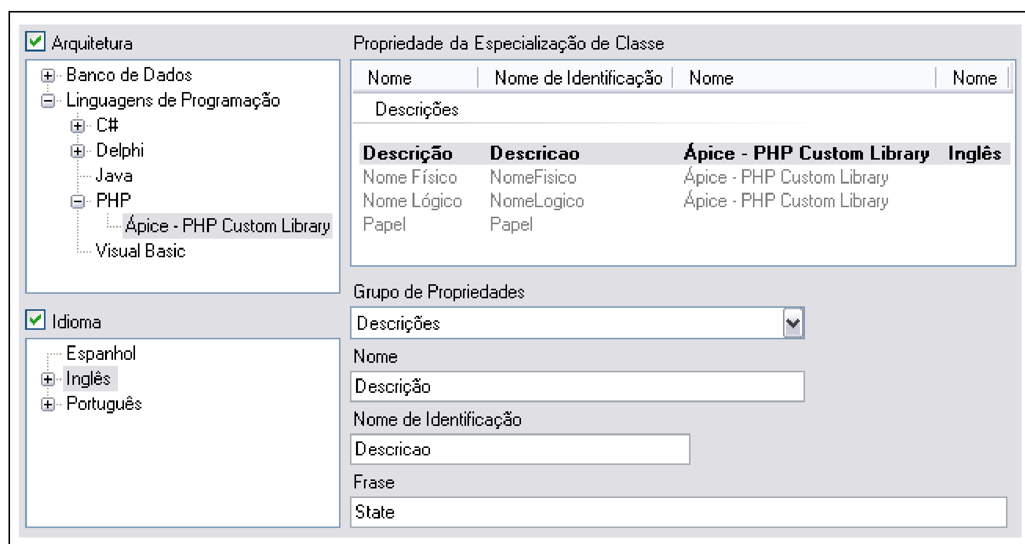


Figura 36 – Especialização da propriedade "Descrição" para o idioma inglês

O Quadro 13 apresenta um trecho do código gerado para a classe "Estado" utilizando o idioma português. Nota-se em destaque a utilização da propriedade "Descrição" anteriormente citada. O exemplo completo da geração de código para a arquitetura PCL encontra-se no Anexo B.

```
class Estado extends CustomEntity {

    var $attributes = NULL;
    var $views = NULL;
    var $relations = NULL;

    function Estado($owner) {
        $this->CustomEntity($owner);

        $this->setPhysicalName("tb_est");
        $this->setLogicalName("estado");
        $this->setDescription("Estado");
    }

    function estadosPorPais($identificadorPais) {
        return NULL;
    }

    function getAttributes() {
        if ($this->attributes == NULL)
            $this->attributes = new EstadoAtributos($this);

        return $this->attributes;
    }
```

Quadro 13 – Trecho do código PCL gerado para a classe “Estado”

Além das classes de entidades utilizadas neste exemplo, o *plugin* “QuelleGenerationPCL” também realiza a geração de classes de interface permitindo a criação de um software completo para inclusão, alteração e exclusão de dados, considerando também os relacionamentos entre as classes. Um exemplo de interface gerada utilizando o idioma inglês é apresentada na Figura 37. Os demais exemplos de interfaces geradas encontram-se no Anexo C.

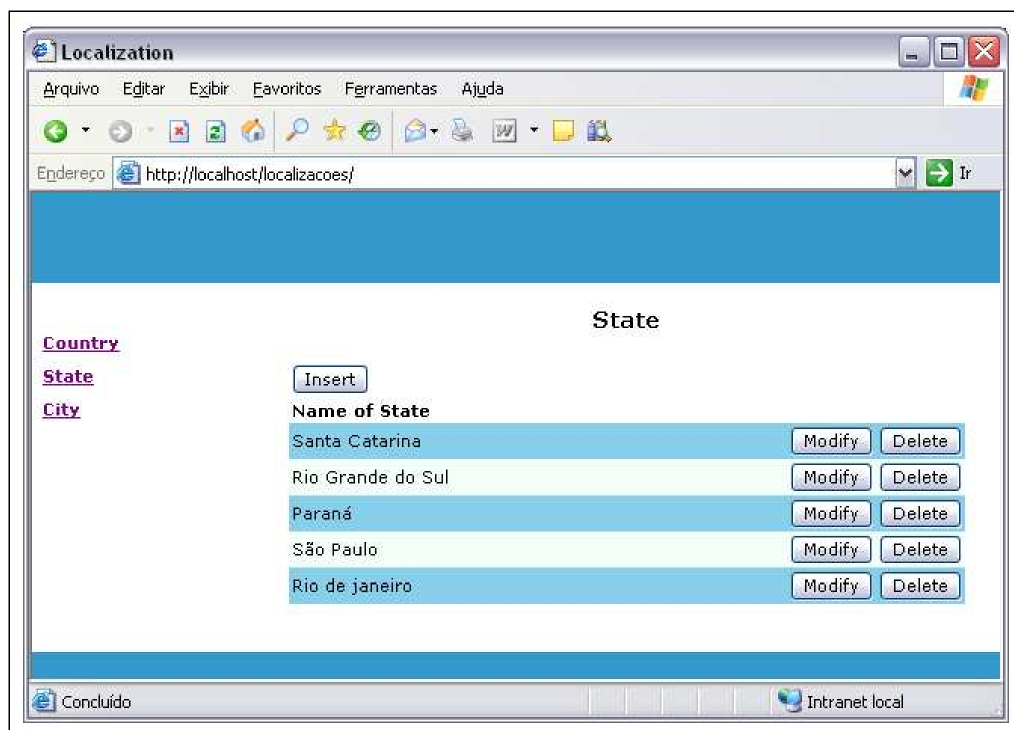


Figura 37 – Exemplo de interface PCL gerada para manutenção da classe “Estado”

3.4 RESULTADOS E DISCUSSÃO

Durante a fase inicial do trabalho identificou-se uma grande preocupação em relação ao tempo disponível para o desenvolvimento do mesmo, devido ao grande número de tecnologias utilizadas, dentre elas pode se citar o Microsoft .NET, Visual C#, XML (*Extensible Markup Language*), XMI (*XML Metadata Interchange*) e UML (*Unified Modeling Language*), além das ferramentas utilizadas tanto na fase de análise como na fase de implementação da ferramenta. O estudo das tecnologias e ferramentas citadas consumiu uma grande parcela do tempo destinado ao desenvolvimento do trabalho.

Ao se comparar a ferramenta proposta com as ferramentas destinadas ao processo de desenvolvimento de software disponíveis no mercado, nota-se que a mesma apresenta uma estrutura de especialização não encontrada nas demais. Pela estrutura de especialização utilizada e por ser uma ferramenta extremamente técnica, foram utilizados durante o desenvolvimento do trabalho vários exemplos com a finalidade de facilitar a compreensão da mesma.

Na fase de construção da Fundamentação Teórica as maiores dificuldades encontradas foram as diferentes maneiras que os autores de livros UML citados descrevem um problema. Um exemplo desta dificuldade é o livro de Medeiros (2004). Neste livro muitos termos definidos pela UML são substituídos por termos utilizados pela linguagem de programação Java, o que dificulta e muito a compreensão do leitor.

Todo o estudo realizado na Fundamentação Teórica demonstrou-se de suma importância na fase de implementação da ferramenta, em específico os estudos realizados sobre o diagrama de classes UML. O conhecimento do diagrama de classes UML e de todos os seus elementos (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros e Relacionamentos) são um pré-requisito fundamental para a utilização da tecnologia XMI.

A utilização de diagramas de classes UML e suas especializações propostas pelo trabalho fornecem ao desenvolvedor um repositório rico em informações para a geração de código para diferentes arquiteturas e idiomas. A preocupação de utilizar um mesmo repositório de dados para gerar código em diferentes arquiteturas também é perceptível nos trabalhos correlatos de Schmidt (2001) e Wehrmeister (2001).

A possibilidade de gerar código para diferentes arquiteturas proposta pelo trabalho agrega ao processo de desenvolvimento uma maior padronização na criação dos modelos de dados, não havendo a necessidade da construção de um diagrama de classes de implementação para cada arquitetura em especial, amenizando desta forma as mudanças necessárias para a utilização de uma nova tecnologia, permitindo também a construção de modelos em um nível mais elevado.

Conforme demonstrado no tópico 3.3.3.1 (Geração de Código e *Plugins*) a utilização de um repositório de dados rico em informações permite a construção de *plugins* para geração de código mais refinados, de forma a atender as particularidades de cada arquitetura, sendo que esta flexibilidade não é encontrada nas ferramentas CASE existentes no mercado.

A utilização da tecnologia Microsoft .NET foi considerada um pré-requisito para o desenvolvimento do trabalho pois a mesma proporciona enorme flexibilidade para a construção dos *plugins* de geração de código. Uma das suas principais características é o suporte as principais linguagens de programação existentes no mercado como Java, Visual Basic, Delphi e Visual C#. Está característica permite que o desenvolvedor construa seus *plugins* de geração utilizando a linguagem de programação mais próxima da sua realidade.

A construção dos *plugins* através de módulos DLL (*Dynamic Link Library*) proporciona ao desenvolvedor uma maior flexibilidade durante a fase de implementação, pois os mesmos utilizam as funcionalidades e subsídios do Microsoft .NET. Outro fator considerável é o aumento de velocidade no processo de geração de código, pois ao contrário dos *scripts* utilizados pelas ferramentas CASE existentes no mercado os mesmos são compilados e não interpretados.

4 CONCLUSÕES

O objetivo do trabalho foi atingido, o qual consistia no desenvolvimento de uma ferramenta de auxílio ao processo de desenvolvimento que possibilitasse a geração de código com base na especialização de diagramas de classes UML (*Unified Modeling Language*) por arquitetura e idioma. Para o desenvolvimento da ferramenta foram utilizadas tecnologias e ferramentas conceituadas no mercado de desenvolvimento de software. Algumas tecnologias como o Microsoft .NET e XMI (*XML Metadata Interchange*) foram consideradas pré-requisitos para a implementação da mesma.

Todos os objetivos específicos do trabalho também foram atingidos. A importação dos diagramas de classes ocorre através de artefatos XMI. Os diagramas de classes importados são especializados por arquitetura e idioma através de uma interface gráfica única, o que facilita a especialização de cada elemento destes diagramas. A interface para acoplamento de *plugins* disponibiliza aos mesmos os diagramas de classes especializados através de coleções, facilitando assim o acesso aos dados para implementação da lógica de geração.

A melhora na qualidade do software desenvolvido e a redução do tempo de desenvolvimento abordados na introdução do trabalho, não são mais vistos pelo mercado como um diferencial, mas sim como um requisito fundamental para fechamento de negócios. Com base nos problemas citados, o processo de Engenharia da Produção (*Forward Engineering*) através da geração de código tem a função de amenizar estes problemas. A geração de código reduz o tempo do processo de desenvolvimento, pois minimiza a necessidade de codificação manual e aumenta a confiabilidade do código gerado, pois é produzido por uma ferramenta depurada e testada (FISHER, 1990, p.10).

Além do aumento da qualidade e redução do tempo de desenvolvimento, o presente trabalho ajuda a agregar uma melhor padronização ao processo de desenvolvimento, pois o mesmo requer o uso da linguagem de documentação e modelagem UML e de metodologias propostas pela Engenharia de Software. Nota-se também que um processo de desenvolvimento de software bem definido é um grande passo para a obtenção de certificações através dos Sistemas de Garantia da Qualidade.

A estrutura de especialização através da utilização de propriedades genéricas permite a abstração de dados comuns para as diferentes arquiteturas utilizadas, possibilitando desta

forma o reaproveitamento de informações e enriquecimento do repositório de dados da ferramenta. A especialização por arquitetura permite o isolamento de todas as características de uma determinada arquitetura, mantendo desta forma um repositório de dados mais limpo e organizado, facilitando o uso e a manutenção da ferramenta e tornando os softwares gerados mais flexíveis perante as mudanças tecnológicas. A especialização por idioma fornece um recurso fundamental para as organizações na internacionalização de seus softwares, pois através da mesma é possível realizar a geração de código para diferentes idiomas.

A utilização de diagramas de classes especializados permite a construção de *plugins* para geração de código mais inteligentes e com a capacidade de abstrair a utilização de um diagrama de classes de implementação. A abstração citada permite ao analista um foco maior no negócio em questão, poupando também o esforço gasto pela equipe de desenvolvimento na implementação de rotinas básicas.

A ferramenta desenvolvida apresenta algumas deficiências ou limitações, principalmente na importação dos diagramas de classes, pois utiliza somente o formato XMI. Além desta limitação não é possível através da ferramenta a criação de fórmulas e dados compostos.

4.1 EXTENSÕES

As sugestões para trabalhos futuros podem ser divididas em três categorias: entrada de dados, saída de dados e especialização.

Na categoria entrada de dados a sugestão é a pesquisa e implementação de rotinas que proporcionem a importação dos diagramas de classes UML (*Unified Modeling Language*) a partir de formatos proprietários das ferramentas CASE (*Computer Aided Software Engineering*) existentes no mercado. Atualmente a ferramenta realiza a importação destes diagramas através de artefatos no formato XMI (*XML Metadata Interchange*) versão 1.2. Outra sugestão é a implementação da importação utilizando outras versões do formato XMI.

Na categoria saída de dados a sugestão é a construção de *plugins* para geração de código ou documentação. No quesito geração de código a sugestão é criar *plugins* que atendam as tendências atuais do mercado, tanto para linguagens de programação como também para banco de dados. No quesito documentação a sugestão é criar *plugins* que atendam as normas de documentação propostas pelos Sistemas de Garantia da Qualidade

como CMM (*Capability Maturity Model*) e ISO (*International Organization for Standardization*) tomando como base os diagramas de classes especializados. Outra sugestão sobre a geração de código é a criação de *plugins* que permitam alterações no código gerado, facilitando desta forma a manutenção do software em questão.

A categoria de especialização está dividida em duas sugestões: a primeira sugere a construção de novos tipos de dado que possibilitem a ferramenta a geração de regras de negócio juntamente com a utilização de fórmulas. A segunda sugestão é a criação de propriedades compostas, conforme demonstrado na Tabela 9.

Tabela 9 – Propriedades compostas

Propriedades	**Composições**	**Valores**
Prefixo		“Cls”
Nome Lógico		“Estado”
Nome de Declaração	<Prefixo> + <Nome Lógico>	“ClsEstado”

REFERÊNCIAS BIBLIOGRÁFICAS

ANDERSON, Richard et al. **Professional XML**. Rio de Janeiro: Ciência Moderna, 2001. 1266 p.

ARAÚJO, Eratóstenes Edson Ramalho de. A internacionalização e a localização de produtos e serviços: a sua importância na indústria de software, **T&C Amazônia**. [S.l.], n. 2, p. 44-48, jun. 2003.

BALLMANN, Vanderlei. **Protótipo de ferramenta CASE para geração de código C++ e diagrama de classes**. 2000. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação). Universidade Regional de Blumenau. Blumenau, 2000.

BEZZERA, Eduardo. **Princípios da análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2003. 286 p.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML, guia do usuário**. Rio de Janeiro: Campus, 2000. 472 p.

FISHER, Alan S. **CASE**: utilização de ferramentas para desenvolvimento de software. Rio de Janeiro: Campus, 1990. 264 p.

FURLAN, José Davi. **Modelagem de objetos através da UML**. São Paulo: Makron Books, 1998. 329 p.

GUEDES, Gilleanes T. A. **UML, uma abordagem prática**. São Paulo: Novatec, 2004. 319 p.

KRAMEL, Danilo. **Protótipo de software para geração de código CDL através do repositório da ferramenta CASE System Architect**. 2000. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação). Universidade Regional de Blumenau. Blumenau, 2000.

LIONBRIDGE, Technologies. **Internacionalização de software**. [S.l.], 2005. Disponível em: <http://www.lionbridge.fr/globalization/capabilities/software-internationalization.liox?intLangID=10>. Acesso em: 13 jul. 2005.

MARTIN, James. **Princípios de análise e projeto baseados em objetos**. 4. ed. Rio de Janeiro: Campus, 1994. 486 p.

MEDEIROS, Ernani Sales. **UML 2.0 definitivo**. São Paulo: Pearson Makron Books, 2004. 264p.

OBJECT MANAGEMENT GROUP. **OMG XML Metadata Interchange (XMI) Specification**. [S.l.], 2002. Disponível em: < http://www.omg.org/cgi-bin/apps/doc?formal/02-01-01.pdf >. Acesso em: 30 ago. 2004.

OBJECT MANAGEMENT GROUP. **UML 2.0 Superstructure Specification**. [S.l.], 2004. Disponível em: <http://www.omg.org/docs/ptc/04-10-02.pdf>. Acesso em: 25 mar. 2005.

PAULA FILHO, Wilson de Pádua. **Engenharia de software:** fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2003. 602 p.

PFLEEGER, Shari Lawrence. **Engenharia de software:** teoria e prática. 2. ed. São Paulo: Pearson Makron Books, 2004. 535 p.

REZENDE, Denis Alcides. **Engenharia de software e sistemas de informações**. Rio de Janeiro: Brasport, 1999. 324 p.

SCHMIDT, Roger Anderson. **Ferramenta de auxílio ao processo de desenvolvimento de software integrando tecnologias otimizadoras**. 2001. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação).Universidade Regional de Blumenau. Blumenau, 2001.

SILVEIRA, Claudionor. **Geração automática de cadastros e consultas para linguagem ASP baseado em banco de dados**. 2003. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação). Universidade Regional de Blumenau. Blumenau, 2003.

WEHRMEISTER, Marco Aurélio. **Software para geração de código fonte a partir do repositório da ferramenta CASE System Architect**. 2001. Monografia (Conclusão de Curso de Bacharel em Ciências da Computação). Universidade Regional de Blumenau. Blumenau, 2001.

APÊNDICE A – Descrição dos casos de uso

CSU 01.01 – MANTER PROJETOS

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados dos projetos.

Ator Primário: Usuário

Ator Secundário: Artefato XMI

Fluxo Principal

1. O usuário requisita a manutenção de projetos.
2. A ferramenta lista os projetos já cadastrados através de uma lista de navegação.
3. A ferramenta apresenta a opção para inserir projeto.
4. O usuário opta por inserir um projeto, navegar pelos projetos listados ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar um projeto na lista de navegação

a. O usuário seleciona um projeto na lista de navegação.

b. A ferramenta apresenta os dados do projeto selecionado (nome, *path* do artefato XMI que contém o modelo de classes do projeto) e libera os mesmos para alteração.

c. A ferramenta apresenta a opção para excluir o projeto selecionado.

d. Caso o usuário altere algum dado do projeto a ferramenta apresenta opções para cancelar o projeto alterado, cancelar todos os projetos modificados, salvar o projeto alterado ou salvar todos os projetos modificados.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir um projeto.

b. A ferramenta apresenta um formulário em branco com os dados do projeto para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar o projeto inserido, cancelar todos os projetos modificados, salvar o projeto inserido ou salvar todos os projetos modificados.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações do projeto selecionado.

b. Caso o projeto selecionado esteja em estado de inserção a ferramenta cancela a inclusão do mesmo; caso o projeto selecionado esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os projetos inseridos, alterados ou removidos, caso o projeto esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta o projeto removido na lista de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações do projeto selecionado.

b. A ferramenta realiza o salvamento físico do projeto selecionado.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os projetos inseridos, alterados ou removidos, caso o projeto esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física do projeto e de todas as suas especializações.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção do projeto selecionado.

b. A ferramenta realiza a remoção lógica do projeto, removendo também o mesmo da lista de navegação e apresenta opções para cancelar todos os projetos modificados, ou salvar todos os projetos modificados.

Pós-condição: Um projeto foi inserido, removido ou seus dados foram alterados.

CSU 01.02 – MANTER ARQUITETURAS

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados das arquiteturas.

Ator Primário: Usuário

Ator Secundário: *Plugin* para geração de código

Fluxo Principal

1. O usuário requisita a manutenção de arquiteturas.
2. A ferramenta lista as arquiteturas já cadastradas através de uma árvore de navegação.
3. A ferramenta apresenta a opção para inserir arquitetura.
4. O usuário opta por inserir uma arquitetura, navegar pelas arquiteturas listadas ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar uma arquitetura na árvore de navegação

a. O usuário seleciona uma arquitetura na árvore de navegação.

b. A ferramenta apresenta os dados da arquitetura selecionada (nome e *plugin* para geração de código) e libera os mesmos para alteração.

c. A ferramenta apresenta as opções para inserir uma arquitetura descendente ou excluir a arquitetura selecionada.

d. Caso o usuário altere algum dado da arquitetura a ferramenta apresenta opções para cancelar a arquitetura alterada, cancelar todas as arquiteturas modificadas, salvar a arquitetura alterada ou salvar todas as arquiteturas modificadas.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir uma arquitetura genérica.

b. A ferramenta apresenta um formulário em branco com os dados da arquitetura para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar a arquitetura inserida, cancelar todas as arquiteturas modificadas, salvar a arquitetura inserida ou salvar todas as arquiteturas modificadas.

Fluxo Alternativo: Inserir Descendente

a. O usuário opta por inserir uma arquitetura descendente a arquitetura selecionada.

b. A ferramenta apresenta um formulário em branco com os dados da arquitetura para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar a arquitetura inserida, cancelar todas as arquiteturas modificadas, salvar a arquitetura inserida ou salvar todas as arquiteturas modificadas.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações da arquitetura selecionada.

b. Caso a arquitetura selecionada esteja em estado de inserção a ferramenta cancela a inclusão da mesma; caso a arquitetura selecionada esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as arquiteturas inseridas, alteradas ou removidas, caso a arquitetura esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta a arquitetura removida na árvore de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações da arquitetura selecionada.

b. A ferramenta realiza o salvamento físico da arquitetura selecionada.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as arquiteturas inseridas, alteradas ou removidas, caso a arquitetura esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física da arquitetura.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção da arquitetura selecionada juntamente com suas arquiteturas descendentes.

b. Se a arquitetura pode ser removida, a ferramenta realiza a remoção lógica da arquitetura, removendo a mesma da árvore de navegação e apresenta opções para cancelar todas as arquiteturas modificadas, ou salvar todas as arquiteturas modificadas; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **A arquitetura não pode ser removida**

a. A arquitetura já está sendo utilizada por uma ou mais especializações.

b. A ferramenta reporta o fato retornando ao Fluxo Principal.

Pós-condição: Uma arquitetura foi inserida, removida ou seus dados foram alterados.

CSU 01.03 – MANTER IDIOMAS

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados dos idiomas.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção de idiomas.
2. A ferramenta lista os idiomas já cadastrados através de uma árvore de navegação.
3. A ferramenta apresenta a opção para inserir idioma.
4. O usuário opta por inserir um idioma, navegar pelos idiomas listados ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar um idioma na árvore de navegação

a. O usuário seleciona um idioma na árvore de navegação.

b. A ferramenta apresenta os dados do idioma selecionado (nome) e libera os mesmos para alteração.

c. A ferramenta apresenta as opções para inserir um idioma descendente ou excluir o idioma selecionado.

d. Caso o usuário altere algum dado do idioma a ferramenta apresenta opções para cancelar o idioma alterado, cancelar todos os idiomas modificados, salvar o idioma alterado ou salvar todos os idiomas modificados.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir um idioma genérico.

b. A ferramenta apresenta um formulário em branco com os dados do idioma para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar o idioma inserido, cancelar todos os idiomas modificados, salvar o idioma inserido ou salvar todos os idiomas modificados.

Fluxo Alternativo: Inserir Descendente

a. O usuário opta por inserir um idioma descendente ao idioma selecionado.

b. A ferramenta apresenta um formulário em branco com os dados do idioma para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar o idioma inserido, cancelar todos os idiomas modificados, salvar o idioma inserido ou salvar todos os idiomas modificados.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações do idioma selecionado.

b. Caso o idioma selecionado esteja em estado de inserção a ferramenta cancela a inclusão do mesmo; caso o idioma selecionado esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os idiomas inseridos, alterados ou removidos, caso o idioma esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta o idioma removido na árvore de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações do idioma selecionado.

b. A ferramenta realiza o salvamento físico do idioma selecionado.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os idiomas inseridos, alterados ou removidos, caso o idioma esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física do idioma.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção do idioma selecionado juntamente com seus idiomas descendentes.

b. Se o idioma pode ser removido, a ferramenta realiza a remoção lógica do idioma, removendo também o mesmo da árvore de navegação e apresenta opções para cancelar todos os idiomas modificados, ou salvar todos os idiomas modificados; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **O idioma não pode ser removido**

a. O idioma já está sendo utilizada por uma ou mais especializações.

b. A ferramenta reporta o fato retornando ao Fluxo Principal.

Pós-condição: Um idioma foi inserido, removido ou seus dados foram alterados.

CSU 01.04 – MANTER TIPOS DE DADO

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados dos tipos de dado.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção dos tipos de dado.

2. A ferramenta lista os tipos de dado já cadastrados através de uma árvore de navegação.

3. A ferramenta apresenta a opção para inserir tipo de dado.

4. O usuário opta por inserir um tipo de dado, navegar pelos tipos de dado listados ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar um tipo de dado na árvore de navegação

a. O usuário seleciona um tipo de dado na árvore de navegação.

b. A ferramenta apresenta os dados do tipo de dado selecionado (nome, nome de identificação) e libera os mesmos para alteração.

c. A ferramenta habilita o caso de uso estendido 01.05 (Manter Propriedades de Tipo de Dado), passando como parâmetro o tipo de dado selecionado, para que o usuário possa realizar a manutenção das propriedades do tipo de dado.

d. A ferramenta apresenta as opções para inserir um tipo de dado descendente ou excluir o tipo de dado selecionado.

e. Caso o usuário altere algum dado do tipo de dado a ferramenta apresenta opções para cancelar o tipo de dado alterado, cancelar todos os tipos de dado modificados, salvar o tipo de dado alterado ou salvar todos os tipos de dado modificados.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir um tipo de dado genérico.

b. A ferramenta apresenta um formulário em branco com os dados do tipo de dado para realização do preenchimento pelo usuário.

c. A ferramenta habilita o caso de uso estendido 01.05 (Manter Propriedades de Tipo de Dado), passando como parâmetro o tipo de dado inserido, para que o usuário possa realizar a manutenção das propriedades do tipo de dado.

d. A ferramenta apresenta opções para cancelar o tipo de dado inserido, cancelar todos os tipos de dado modificados, salvar o tipo de dado inserido ou salvar todos os tipos de dado modificados.

Fluxo Alternativo: Inserir Descendente

a. O usuário opta por inserir um tipo de dado descendente ao tipo de dado selecionado.

b. A ferramenta apresenta um formulário em branco com os dados do tipo de dado para realização do preenchimento pelo usuário.

c. A ferramenta habilita o caso de uso estendido 01.05 (Manter Propriedades de Tipo de Dado), passando como parâmetro o tipo de dado inserido, para que o usuário possa realizar a manutenção das propriedades do tipo de dado.

d. A ferramenta apresenta opções para cancelar o tipo de dado inserido, cancelar todos os tipos de dado modificados, salvar o tipo de dado inserido ou salvar todos os tipos de dado modificados.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações do tipo de dado selecionado.

b. Caso o tipo de dado selecionado esteja em estado de inserção a ferramenta cancela a inclusão do mesmo, desabilitando também o caso de uso estendido 01.05 (Manter Propriedades de Tipo de Dado); caso o tipo de dado selecionado esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os tipos de dado inseridos, alterados ou removidos, caso o tipo de dado esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta o tipo de dado removido na árvore de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações do tipo de dado selecionado.

b. A ferramenta realiza o salvamento físico do tipo de dado selecionado.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os tipos de dado inseridos, alterados ou removidos, caso o tipo de dado esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física do tipo de dado e de todas as suas propriedades de tipo de dado.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção do tipo de dado selecionado juntamente com seus tipos de dado descendentes.

b. Se o tipo de dado pode ser removido, a ferramenta realiza a remoção lógica do tipo de dado, removendo o mesmo da árvore de navegação, desabilitando também o caso de uso estendido 01.05 (Manter Propriedades de Tipo de Dado) e apresenta opções para cancelar todos os tipos de dado modificados, ou salvar todos os tipos de dado modificados; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **O tipo de dado não pode ser removido**

a. O tipo de dado já está sendo utilizado por um ou mais modelo de classes.

b. A ferramenta reporta o fato retornando ao Fluxo Principal.

Pós-condição: Um tipo de dado foi inserido, removido, ou seus dados foram alterados.

CSU 01.05 – MANTER PROPRIEDADES DE TIPO DE DADO

Sumário: O usuário realiza a manutenção (inclusão, sobreposição, remoção, alteração) dos dados das propriedades de tipo de dado.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção das propriedades de tipo de dado.

2. A ferramenta filtra as propriedades de tipo de dado conforme o tipo de dado indicado pelo caso de uso ao qual o caso de uso atual é estendido.

3. A ferramenta apresenta opções de filtro por arquitetura e idioma.

4. A ferramenta lista as propriedades de tipo de dado já cadastradas conforme o tipo de dado, arquitetura e idioma filtrado, as propriedades de tipo de dado listadas podem pertencer ao tipo de dado atualmente filtrado ou a seus tipos de dado genéricos, sendo as mesmas destacadas visualmente.

5. A ferramenta apresenta opções para inserir as propriedades de tipo de dado, que podem ser do tipo: frase, texto, número inteiro, número decimal, lógico, data, hora, lista de seleção e lista de múltipla seleção.

6. O usuário opta por inserir uma propriedade de tipo de dado, navegar pelas propriedades de tipo de dado listadas ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar uma propriedade de tipo de dado na lista de navegação

a. O usuário seleciona uma propriedade de tipo de dado na lista de navegação.

b. A ferramenta apresenta os dados da propriedade de tipo de dado selecionada (grupo de propriedades, nome, nome de identificação e os dados referentes ao seu tipo); caso a propriedade de tipo de dado selecionada pertença a algum tipo de dado genérico, a ferramenta disponibiliza a opção para sobrescrever a mesma para o tipo de dado atualmente filtrado, não liberando seus dados para a alteração; caso a propriedade de tipo de dado selecionada pertença ao tipo de dado atualmente filtrado a ferramenta libera suas informações para alteração e disponibiliza a opção para excluir a mesma.

c. Caso o usuário altere algum dado da propriedade de tipo de dado a ferramenta apresenta opções para cancelar a propriedade de tipo de dado alterada, cancelar todas as propriedades de tipo de dado modificadas, salvar a propriedade de tipo de dado alterada ou salvar todas as propriedades de tipo de dado modificadas.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir uma propriedade de tipo de dado.

b. A ferramenta lista através de um *menu* os tipos de propriedades de tipo de dado.

c. O usuário escolhe o tipo da propriedade de tipo de dado.

d. A ferramenta apresenta um formulário em branco com os dados da propriedade de tipo de dado para realização do preenchimento pelo usuário.

e. A ferramenta apresenta opções para cancelar a propriedade de tipo de dado inserida, cancelar todas as propriedades de tipo de dado modificadas, salvar a propriedade de tipo de dado inserida ou salvar todas as propriedades de tipo de dado modificadas.

Fluxo Alternativo: Sobrescrever

a. O usuário opta por sobrescrever uma propriedade de tipo de dado.

b. A ferramenta insere uma propriedade de tipo de dado com base na propriedade de tipo de dado atualmente selecionada, levando em consideração o filtro de arquitetura e idioma.

c. A ferramenta apresenta um formulário com os dados da propriedade de tipo de dado para realização do preenchimento complementar pelo usuário.

d. A ferramenta apresenta opções para cancelar a propriedade de tipo de dado sobrescrita, cancelar todas as propriedades de tipo de dado modificadas, salvar a propriedade de tipo de dado sobrescrita ou salvar todas as propriedades de tipo de dado modificadas.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações da propriedade de tipo de dado selecionada.

b. Caso a propriedade de tipo de dado selecionada esteja em estado de inserção a ferramenta cancela a inclusão da mesma; caso a propriedade de tipo de dado selecionada esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as propriedades de tipo de dado inseridas, alteradas ou removidas, caso a propriedade de tipo de dado esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta a propriedade de tipo de dado removida na lista de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações da propriedade de tipo de dado selecionada.

b. A ferramenta realiza o salvamento físico da propriedade de tipo de dado selecionada.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as propriedades de tipo de dado inseridas, alteradas ou removidas, caso a propriedade de tipo de dado esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física da propriedade de tipo de dado.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção da propriedade de tipo de dado selecionada.

b. A ferramenta realiza a remoção lógica da propriedade de tipo de dado, removendo também a mesma da lista de navegação e apresenta opções para cancelar todas as

propriedades de tipo de dado modificadas, ou salvar todas as propriedades de tipo de dado modificadas; caso contrário a ferramenta reporta o fato.

Pré-condição: O caso de uso ao qual o caso de uso atual foi estendido forneça o filtro indicando o tipo de dado ao qual as propriedades de tipo de dado listadas pertença.

Pós-condição: Uma propriedade de tipo de dado foi inserida, sobrescrita, removida ou seus dados foram alterados.

CSU 01.06 – MANTER GRUPOS DE PROPRIEDADES

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados dos grupos de propriedades.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção de grupos de propriedades.
2. A ferramenta lista os grupos de propriedades já cadastrados através de uma lista de navegação.
3. A ferramenta apresenta a opção para inserir grupo de propriedades.
4. O usuário opta por inserir um grupo de propriedades, navegar pelos grupos de propriedades listados ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar um grupo de propriedades na lista de navegação

a. O usuário seleciona um grupo de propriedades na lista de navegação.

b. A ferramenta apresenta os dados do grupo de propriedades selecionado (nome) e libera os mesmos para alteração.

c. A ferramenta apresenta a opção para excluir o grupo de propriedades selecionado.

d. Caso o usuário altere algum dado do grupo de propriedades a ferramenta apresenta opções para cancelar o grupo de propriedades alterado, cancelar todos os grupos de propriedades modificados, salvar o grupo de propriedades alterado ou salvar todos os grupos de propriedades modificados.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir um grupo de propriedades.

b. A ferramenta apresenta um formulário em branco com os dados do grupo de propriedades para realização do preenchimento pelo usuário.

c. A ferramenta apresenta opções para cancelar o grupo de propriedades inserido, cancelar todos os grupos de propriedades modificados, salvar o grupo de propriedades inserido ou salvar todos os grupos de propriedades modificados.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações do grupo de propriedades selecionado.

b. Caso o grupo de propriedades selecionado esteja em estado de inserção a ferramenta cancela a inclusão do mesmo; caso o grupo de propriedades selecionado esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os grupos de propriedades inseridos, alterados ou removidos, caso o grupo de propriedades esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta o grupo de propriedades removido na lista de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações do grupo de propriedades selecionado.

b. A ferramenta realiza o salvamento físico do grupo de propriedades selecionado.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todos os grupos de propriedades inseridos, alterados ou removidos, caso o grupo de propriedades esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física do grupo de propriedades.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção do grupo de propriedades selecionado.

b. Se o grupo de propriedades pode ser removido, a ferramenta realiza a remoção lógica do grupo de propriedades, removendo também o mesmo da lista de navegação e apresenta opções para cancelar todos os grupos de propriedades modificados, ou salvar todos os grupos de propriedades modificados; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **O grupo de propriedades não pode ser removido**

a. O grupo de propriedades já esta sendo utilizado por uma ou mais especializações

b. A ferramenta reporta o fato retornando ao Fluxo Principal.

Pós-condição: Um grupo de propriedades foi inserido, removido ou seus dados foram alterados.

CSU 02.01 – MANTER ESPECIALIZAÇÃO

Sumário: O usuário realiza a manutenção (inclusão, remoção, alteração) dos dados das especializações.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção de especializações.
2. A ferramenta lista as especializações já cadastradas através de uma árvore de navegação.
3. A ferramenta apresenta a opção para inserir especialização.
4. O usuário opta por inserir uma especialização, navegar pelas especializações listadas ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar uma especialização na árvore de navegação

a. O usuário seleciona uma especialização na árvore de navegação.

b. A ferramenta apresenta os dados da especialização selecionada (nome, estereótipo) e libera os mesmos para alteração.

c. A ferramenta habilita o caso de uso estendido 02.01 (Manter Propriedades de Especialização), passando como parâmetro a especialização selecionada, para que o usuário possa realizar a manutenção das propriedades da especialização.

d. A ferramenta apresenta as opções para inserir uma especialização descendente ou excluir a especialização selecionada.

e. Caso o usuário altere algum dado da especialização a ferramenta apresenta opções para cancelar a especialização alterada, cancelar todas as especializações modificadas, salvar a especialização alterada ou salvar todas as especializações modificadas.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir uma especialização genérica.

b. A ferramenta apresenta um formulário em branco com os dados da especialização para realização do preenchimento pelo usuário.

c. A ferramenta habilita o caso de uso estendido 02.01 (Manter Propriedades de Especialização), passando como parâmetro a especialização inserida, para que o usuário possa realizar a manutenção das propriedades da especialização.

d. A ferramenta apresenta opções para cancelar a especialização inserida, cancelar todas as especializações modificadas, salvar a especialização inserida ou salvar todas as especializações modificadas.

Fluxo Alternativo: Inserir Descendente

a. O usuário opta por inserir uma especialização descendente a especialização selecionada.

b. A ferramenta apresenta um formulário em branco com os dados da especialização para realização do preenchimento pelo usuário.

c. A ferramenta habilita o caso de uso estendido 02.01 (Manter Propriedades de Especialização), passando como parâmetro a especialização inserida, para que o usuário possa realizar a manutenção das propriedades da especialização.

d. A ferramenta apresenta opções para cancelar a especialização inserida, cancelar todas as especializações modificadas, salvar a especialização inserida ou salvar todas as especializações modificadas.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações da especialização selecionada.

b. Caso a especialização selecionada esteja em estado de inserção a ferramenta cancela a inclusão da mesma, desabilitando também o caso de uso estendido 02.01 (Manter Propriedades de Especialização); caso a especialização selecionada esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as especializações inseridas, alteradas ou removidas, caso a especialização esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta a especialização removida na árvore de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações da especialização selecionada.

b. A ferramenta realiza o salvamento físico da especialização selecionada.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as especializações inseridas, alteradas ou removidas, caso a especialização esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física da especialização e de todas as suas propriedades de especialização.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção da especialização selecionada juntamente com suas especializações descendentes.

b. Se a especialização pode ser removida, a ferramenta realiza a remoção lógica da especialização, removendo a mesma da árvore de navegação, desabilitando também o caso de uso estendido 02.01 (Manter Propriedades de Especialização) e apresenta opções para cancelar todas as especializações modificadas, ou salvar todas as especializações modificadas; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **A especialização não pode ser removida**

a. A especialização já esta sendo utilizada por um ou mais modelo de classes.

b. A ferramenta reporta o fato retornando ao Fluxo Principal.

Pós-condição: Uma especialização foi inserida ou removida, ou seus dados foram alterados.

CSU 02.02 – MANTER PROPRIEDADES DE ESPECIALIZAÇÃO

Sumário: O usuário realiza a manutenção (inclusão, sobreposição, remoção, alteração) dos dados das propriedades de especialização.

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a manutenção das propriedades de especialização.

2. A ferramenta filtra as propriedades de especialização conforme a especialização indicada pelo caso de uso ao qual o caso de uso atual é estendido.

3. A ferramenta apresenta opções de filtro por arquitetura e idioma.

4. A ferramenta lista as propriedades de especialização já cadastradas conforme a especialização, arquitetura e idioma filtrado, as propriedades de especialização listadas podem pertencer à especialização atualmente filtrada ou a suas especializações genéricas, sendo as mesmas destacadas visualmente.

5. A ferramenta apresenta opções para inserir as propriedades de especialização, que podem ser do tipo: frase, texto, número inteiro, número decimal, lógico, data, hora, lista de seleção e lista de múltipla seleção.

6. O usuário opta por inserir uma propriedade de especialização, navegar pelas propriedades de especialização listadas ou por finalizar o caso de uso.

Fluxo Alternativo: Selecionar uma propriedade de especialização na lista de navegação

a. O usuário seleciona uma propriedade de especialização na lista de navegação.

b. A ferramenta apresenta os dados da propriedade de especialização selecionada (grupo de propriedades, nome, nome de identificação e os dados referentes ao seu tipo); caso a propriedade de especialização selecionada pertença a alguma especialização genérica, a ferramenta disponibiliza a opção para sobrescrever a mesma para a especialização atualmente filtrada, não liberando seus dados para a alteração; caso a propriedade de especialização selecionada pertença à especialização atualmente filtrada a ferramenta libera suas informações para alteração e disponibiliza a opção para excluir a mesma.

c. Caso o usuário altere algum dado da propriedade de especialização a ferramenta apresenta opções para cancelar a propriedade de especialização alterada, cancelar todas as propriedades de especialização modificadas, salvar a propriedade de especialização alterada ou salvar todas as propriedades de especialização modificadas.

Fluxo Alternativo: Inserir

a. O usuário opta por inserir uma propriedade de especialização.

b. A ferramenta lista através de um *menu* os tipos de propriedades de especialização.

c. O usuário escolhe o tipo da propriedade de especialização.

d. A ferramenta apresenta um formulário em branco com os dados da propriedade de especialização para realização do preenchimento pelo usuário.

e. A ferramenta apresenta opções para cancelar a propriedade de especialização inserida, cancelar todas as propriedades de especialização modificadas, salvar a propriedade de especialização inserida ou salvar todas as propriedades de especialização modificadas.

Fluxo Alternativo: Sobrescrever

a. O usuário opta por sobrescrever uma propriedade de especialização.

b. A ferramenta insere uma propriedade de especialização com base na propriedade de especialização atualmente selecionada, levando em consideração o filtro de arquitetura e idioma.

c. A ferramenta apresenta um formulário com os dados da propriedade de especialização para realização do preenchimento complementar pelo usuário.

d. A ferramenta apresenta opções para cancelar a propriedade de especialização sobrescrita, cancelar todas as propriedades de especialização modificadas, salvar a propriedade de especialização sobrescrita ou salvar todas as propriedades de especialização modificadas.

Fluxo Alternativo: Cancelar

a. O usuário requisita a ferramenta o cancelamento de todas as modificações da propriedade de especialização selecionada.

b. Caso a propriedade de especialização selecionada esteja em estado de inserção a ferramenta cancela a inclusão da mesma; caso a propriedade de especialização selecionada esteja em estado de alteração a ferramenta recupera os dados anteriores à alteração.

Fluxo Alternativo: Cancelar Todos

a. O usuário requisita a ferramenta o cancelamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as propriedades de especialização inseridas, alteradas ou removidas, caso a propriedade de especialização esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Cancelar; caso esteja no estado de exclusão a ferramenta apresenta a propriedade de especialização removida na lista de navegação.

Fluxo Alternativo: Salvar

a. O usuário requisita a ferramenta o salvamento de todas as modificações da propriedade de especialização selecionada.

b. A ferramenta realiza o salvamento físico da propriedade de especialização selecionada.

Fluxo Alternativo: Salvar Todos

a. O usuário requisita a ferramenta o salvamento de todas as modificações realizadas desde o último salvamento.

b. A ferramenta varre todas as propriedades de especialização inseridas, alteradas ou removidas, caso a propriedade de especialização esteja em estado de inserção ou alteração a ferramenta realiza a chamada do Fluxo Alternativo Salvar; caso esteja no estado de exclusão a ferramenta realiza a remoção física da propriedade de especialização.

Fluxo Alternativo: Excluir

a. O usuário requisita a ferramenta a remoção da propriedade de especialização selecionada.

b. A ferramenta realiza a remoção lógica da propriedade de especialização, removendo também a mesma da lista de navegação e apresenta opções para cancelar todas as propriedades de especialização modificadas, ou salvar todas as propriedades de especialização modificadas; caso contrário a ferramenta reporta o fato.

Pré-condição: O caso de uso ao qual o caso de uso atual foi estendido forneça o filtro indicando a especialização ao qual as propriedades de especialização listadas pertença.

Pós-condição: Uma propriedade de especialização foi inserida, sobrescrita, removida, ou seus dados foram alterados.

CSU 03.01 – IMPORTAR MODELO DE CLASSES

Sumário: A ferramenta realiza a importação do modelo de classes a partir de um artefato XMI (*XML Metadata Interchange*).

Ator Primário: Artefato XMI

Fluxo Principal

1. A ferramenta realiza a chamada para importação passando ao caso de uso o *path* do artefato XMI que contém o modelo de classes.

2. A ferramenta verifica a existência do artefato XMI no *path* informado; caso o artefato XMI exista no *path* informado a ferramenta realiza a carga do mesmo; caso contrário a ferramenta reporta o fato.

3. A ferramenta verifica a versão do artefato XMI, caso a versão seja válida a ferramenta realiza a importação de todos os elementos do modelo de classes (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros, Relacionamentos, Tipos de Dado e Estereótipos) juntamente com seus respectivos atributos; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **O artefato XMI não existe no *path* informado**

a. O *path* do artefato XMI informado é inválido.

b. A ferramenta reporta o fato e finaliza o caso de uso.

Fluxo de Exceção: **A versão do artefato XMI não é válida**

a. A versão do artefato XMI não é igual a 1.2.

b. A ferramenta reporta o fato e finaliza o caso de uso.

Pós-condição: Um modelo de classes foi importado a partir de um artefato XMI.

CSU 03.02 – ESPECIALIZAR MODELO DE CLASSES

Sumário: O usuário realiza a especialização dos elementos do modelo de classes (Pacotes, Interfaces, Classes, Atributos, Operações, Parâmetros e Relacionamentos).

Ator Primário: Usuário

Fluxo Principal

1. O usuário requisita a especialização dos modelos de classes.

2. A ferramenta apresenta ao usuário através de uma lista os projetos disponíveis.
3. O usuário opta por selecionar um projeto listado ou por finalizar o caso de uso.

Fluxo Alternativo: **Selecionar um projeto na lista**

a. O usuário seleciona um projeto na lista.
b. A ferramenta realiza a chamada do caso de uso 03-01 (Importar Modelo de Classes) passando ao mesmo o *path* do artefato XMI correspondente ao projeto selecionado, caso não ocorram exceções na importação do modelo de classes a ferramenta apresenta através de uma árvore de navegação todos os elementos do modelo de classes importado.

Fluxo Alternativo: **Selecionar um elemento do modelo de classes na árvore de navegação**

a. O usuário seleciona um elemento do modelo de classes na árvore de navegação.
b. A ferramenta apresenta através de uma lista todos os atributos do elemento do modelo de classes selecionado para consulta do usuário.
c. A ferramenta verifica se o elemento do modelo de classes selecionado já possui especialização; caso o elemento do modelo de classes não possua especialização a ferramenta realiza a criação da mesma, tomando como base o seu estereótipo, através do estereótipo a ferramenta realiza a ligação da especialização do elemento do modelo de classes com a sua especialização ancestral, herdando desta forma todas as propriedades já informadas na mesma.
d. Caso o elemento do modelo de classes selecionado for um pacote a ferramenta habilita o caso de uso estendido 02.02.01 (Manter Especialização de Pacotes); caso for uma interface a ferramenta habilita o caso de uso estendido 02.02.02 (Manter Especializações de Interfaces); caso for uma classe a ferramenta habilita o caso de uso estendido 02.02.03 (Manter Especialização de Classes); caso for um atributo a ferramenta habilita o caso de uso estendido 02.02.04 (Manter Especialização de Atributos); caso for uma operação a ferramenta habilita o caso de uso estendido 02.02.05 (Manter Especialização de Operações); caso for um parâmetro a ferramenta habilita o caso de uso estendido 02.02.06 (Manter Especialização de Parâmetros); caso for um relacionamento a ferramenta habilita o caso de uso estendido 02.02.07 (Manter Especialização de Relacionamentos).

Pós-condição: Os modelos de classes foram especializados.

CSU 03.03 – GERAR CÓDIGO

Sumário: O usuário informa os elementos do modelo de classes a serem gerados e o *plugin* realiza a execução da geração de código.

Ator Primário: Usuário

Ator Secundário: *Plugin*

Fluxo Principal

1. O usuário requisita a geração de código.
2. A ferramenta apresenta ao usuário através de uma lista os projetos disponíveis.
3. A ferramenta apresenta as opções para escolha da arquitetura e idioma.
4. O usuário opta por selecionar um projeto listado, escolher a arquitetura, escolher o idioma ou por finalizar o caso de uso.

Fluxo Alternativo: **Selecionar um projeto na lista**

a. O usuário seleciona um projeto na lista.

b. A ferramenta realiza a chamada do caso de uso 03-01 (Importar Modelo de Classes) passando ao mesmo o *path* do artefato XMI correspondente ao projeto selecionado, caso não ocorram exceções na importação do modelo de classes a ferramenta apresenta através de uma árvore de seleção os elementos pacote, classes e interfaces do modelo de classes para que o usuário faça a escolha dos elementos do modelo de classes a serem gerados.

c. A ferramenta apresenta a opção para realizar a execução da geração.

Fluxo Alternativo: **Executar a geração de código**

a. O usuário opta por realizar a execução da geração de código.

b. A ferramenta cria uma lista com todos os elementos do modelo de classes selecionados pelo usuário, realizando o filtro das propriedades de especialização de cada elemento por arquitetura e idioma.

c. A ferramenta verifica a existência do *plugin* de geração de código correspondente à arquitetura selecionada, caso o *plugin* exista no *path* informado na arquitetura a ferramenta realiza a carga do mesmo executando o método de geração passando como parâmetro a lista com todos os elementos do modelo de classes selecionados; caso contrário a ferramenta reporta o fato.

Fluxo de Exceção: **O *plugin* para geração de código não existe no *path* informado**

a. O *path* do *plugin* de para geração de código informado é inválido.

b. A ferramenta reporta o fato e retorna ao fluxo principal.

Pós-condição: A execução da geração de código foi realizada.

ANEXO A – Exemplos de código gerado para arquitetura Java

Exemplo de fonte gerado através do módulo "QuelleGenerationJava"

```
/*
Fonte Gerado Automaticamente
Plugin: QuelleGenerationJava
Data: 6/6/2005
Hora: 18:16
*/

public class Estado {

    private int identificador = null;
    private String nome = null;
    private String sigla = null;
    private Pais pais = null;

    public Estado() {
        super();
    }

    public int getIdentificador() {
        return this.identificador;
    }

    public void setIdentificador(int identificador) {
        this.identificador = identificador;
    }

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

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

    public String getSigla() {
        return this.sigla;
    }

    public void setSigla(String sigla) {
        this.sigla = sigla;
    }

    public Pais getPais() {
        return this.pais;
    }

    public void setPais(Pais pais) {
        this.pais = pais;
    }

    public int estadosPorPais(int identificadorPais) {
        return null;
    }


}
```

Exemplo de fonte gerado através da ferramenta Enterprise Architect

```
package Localizacoes.Persistencia;

/**
 * @version 1.0
 * @created 29-jun-2005 11:03:39
 */
public class Estado {

    private int identificador;
    private char nome;
```

```java
    private char sigla;
    public Municipio m_Municipio;

    public Estado(){

    }

    public void finalize() throws Throwable {

    }

    public int getidentificador(){
        return identificador;
    }

    public char getnome(){
        return nome;
    }

    public char getsigla(){
        return sigla;
    }

    /**
     *
     * @param newVal
     */
    public void setidentificador(int newVal){
        identificador = newVal;
    }

    /**
     *
     * @param newVal
     */
    public void setnome(char newVal){
        nome = newVal;
    }

    /**
     *
     * @param newVal
     */
    public void setsigla(char newVal){
        sigla = newVal;
    }

    /**
     *
     * @param identificadorPais
     */
    public int estadosPorPais(int identificadorPais){
        return 0;
    }


}
```

ANEXO B – Exemplo de código PCL (*PHP Custom Library*) gerado

```
<?

    /*
    Fonte Gerado Automaticamente
    Plugin: QuelleGenerationPCL
    Data: 7/6/2005
    Hora: 11:23
    */

    require_once("customentity.php");
    require_once("customentityattributes.php");
    require_once("customentityviews.php");
    require_once("customentityrelations.php");
    require_once("pais.php");

    class EstadoAtributos extends CustomEntityAttributes {

        var $identificador = NULL;
        var $nome = NULL;
        var $sigla = NULL;
        var $pais = NULL;

        function EstadoAtributos($owner) {
            $this->CustomEntityAttributes($owner);

            $this->identificador =
            $this->addAttributeInteger("identificadorEstado",
                                       "id_est",
                                       "Identificador do Estado",
                                       true,
                                       false,
                                       false,
                                       5,
                                       NULL,
                                       NULL);

            $this->nome =
            $this->addAttributeString("nomeEstado",
                                      "nm_est",
                                      "Nome do Estado",
                                      false,
                                      false,
                                      false,
                                      40,
                                      NULL,
                                      NULL);

            $this->sigla =
            $this->addAttributeString("siglaEstado",
                                      "sg_est",
                                      "Sigla do Estado",
                                      false,
                                      false,
                                      true,
                                      2,
                                      NULL,
                                      NULL);

            $this->pais =
            $this->addAttributeInteger("identificadorPais",
                                       "id_pai",
                                       "País",
                                       false,
                                       true,
                                       true,
                                       5,
                                       NULL,
                                       NULL);
        }

        function getIdentificador() {
            return $this->getItem($this->identificador);
```

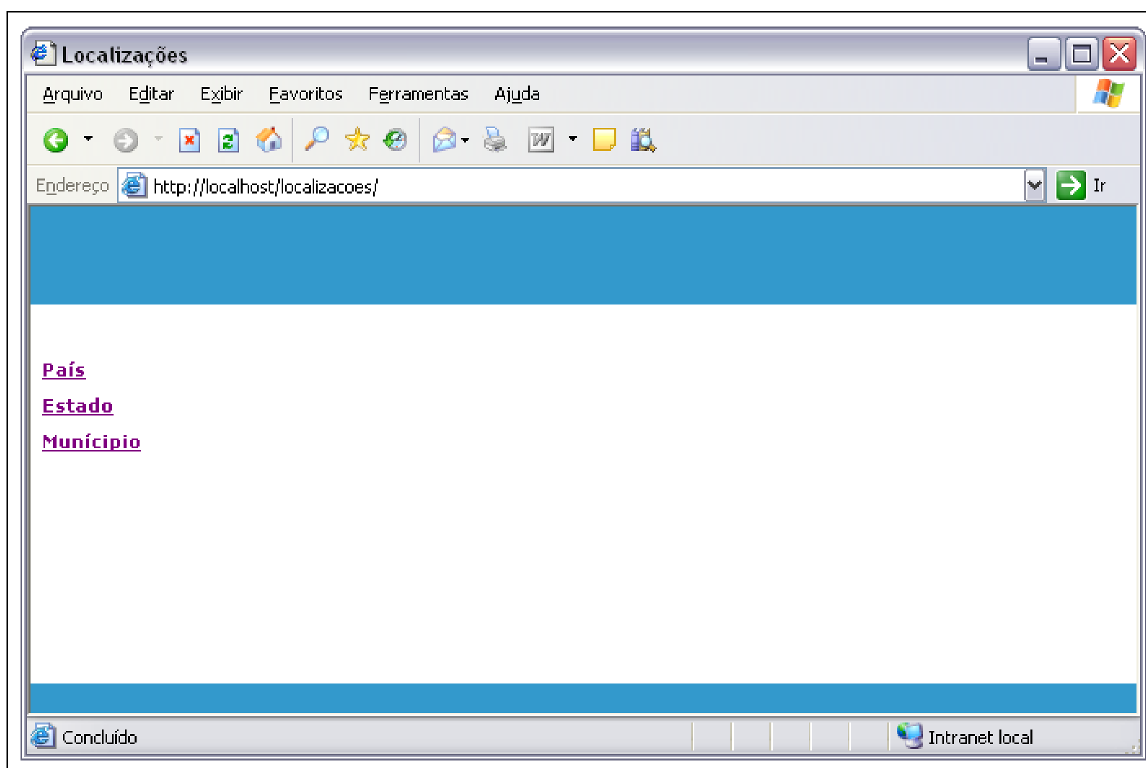
```
    }

    function getNome() {
        return $this->getItem($this->nome);
    }

    function getSigla() {
        return $this->getItem($this->sigla);
    }

    function getPais() {
        return $this->getItem($this->pais);
    }

}

class EstadoVisaoLookup extends CustomEntityView {

    var $identificador = NULL;
    var $nome = NULL;

    function EstadoVisaoLookup($owner) {
        $this->CustomEntityView($owner);

        $this->identificador =
        $this->add($this->getViews()->getEntity()->getAttributes()->getIdentificador(),
                   false);

        $this->nome =
        $this->add($this->getViews()->getEntity()->getAttributes()->getNome(),
                   true);
    }

    function getIdentificador() {
        return $this->getItem($this->identificador);
    }

    function getNome() {
        return $this->getItem($this->nome);
    }

}

class EstadoVisaoLista extends CustomEntityView {

    var $identificador = NULL;
    var $nome = NULL;

    function EstadoVisaoLista($owner) {
        $this->CustomEntityView($owner);

        $this->identificador =
        $this->add($this->getViews()->getEntity()->getAttributes()->getIdentificador(),
                   false);

        $this->nome =
        $this->add($this->getViews()->getEntity()->getAttributes()->getNome(),
                   true);
    }

    function getIdentificador() {
        return $this->getItem($this->identificador);
    }

    function getNome() {
        return $this->getItem($this->nome);
    }

}

class EstadoVisaoManutencao extends CustomEntityView {

    var $identificador = NULL;
    var $nome = NULL;
    var $sigla = NULL;
```

```
    var $pais = NULL;

    function EstadoVisaoManutencao($owner) {
        $this->CustomEntityView($owner);

        $this->identificador =
        $this->add($this->getViews()->getEntity()->getAttributes()->getIdentificador(),
                   false);

        $this->nome =
        $this->add($this->getViews()->getEntity()->getAttributes()->getNome(),
                   true);

        $this->sigla =
        $this->add($this->getViews()->getEntity()->getAttributes()->getSigla(),
                   true);

        $this->pais =
        $this->add($this->getViews()->getEntity()->getAttributes()->getPais(),
                   true);

    }

    function getIdentificador() {
        return $this->getItem($this->identificador);
    }

    function getNome() {
        return $this->getItem($this->nome);
    }

    function getSigla() {
        return $this->getItem($this->sigla);
    }

    function getPais() {
        return $this->getItem($this->pais);
    }

}

class EstadoVisoes extends CustomEntityViews {

    var $lookup = NULL;
    var $lista = NULL;
    var $manutencao = NULL;

    function EstadoVisoes($owner) {

        $item = NULL;

        $this->CustomEntityViews($owner);

        $item = new EstadoVisaoLookup($this);
        $this->lookup = $this->add($item, "lookup", "lookup", "lookup");

        $item = new EstadoVisaoLista($this);
        $this->lista = $this->add($item, "lista", "lista", "lista");

        $item = new EstadoVisaoManutencao($this);
        $this->manutencao = $this->add($item, "manutencao", "manutencao", "manutencao");
    }

    function getLookup() {
        return $this->getItem($this->lookup);
    }

    function getLista() {
        return $this->getItem($this->lista);
    }

    function getManutencao() {
        return $this->getItem($this->manutencao);
    }
```

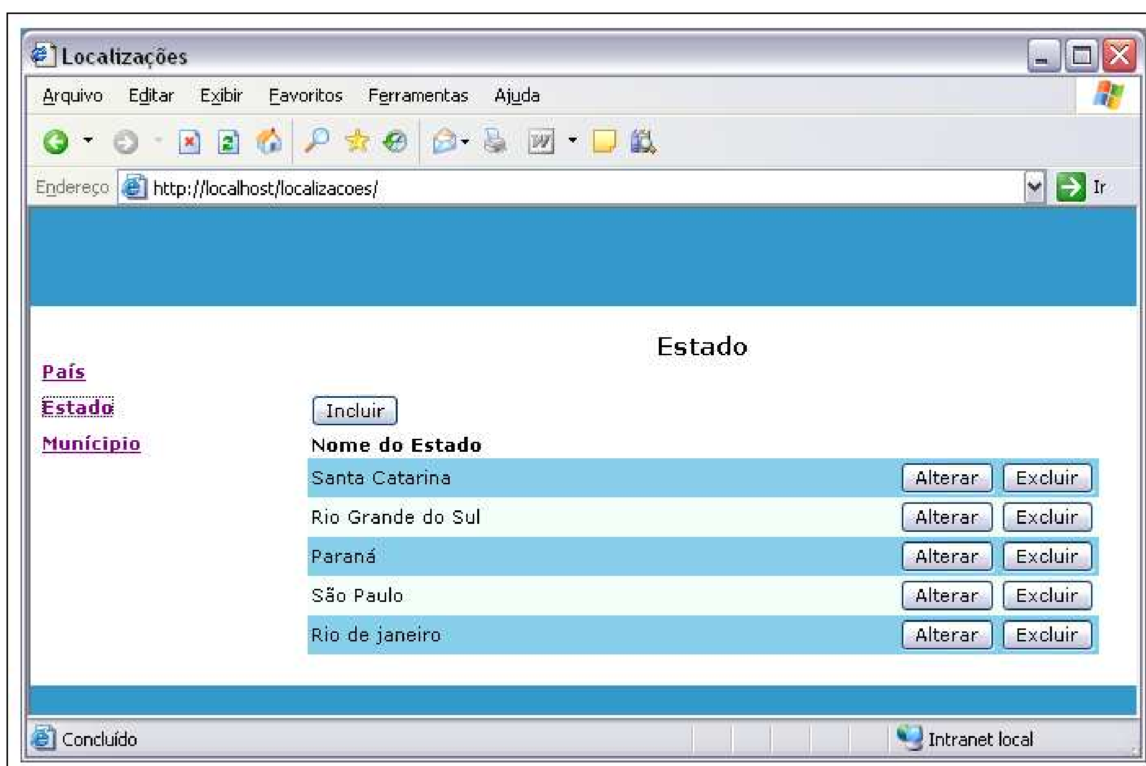
```
    }

    class EstadoRelacionamentos extends CustomEntityRelations {

        var $pais = NULL;

        function EstadoRelacionamentos($owner) {

            $item = NULL;

            $this->CustomEntityRelations($owner);

            $item = new Pais($this);
            $this->pais = $this->add($item,
                                     $item->getAttributes()->getIdentificador(),
                                     $item->getViews()->getLookup(),
                                     $this->getEntity()->getAttributes()->getPais());
        }

        function getPais() {
            return $this->getItem($this->pais);
        }

    }

    class Estado extends CustomEntity {

        var $attributes = NULL;
        var $views = NULL;
        var $relations = NULL;

        function Estado($owner) {
            $this->CustomEntity($owner);

            $this->setPhysicalName("tb_est");
            $this->setLogicalName("estado");
            $this->setDescription("Estado");
        }

        function estadosPorPais($identificadorPais) {
            return NULL;
        }

        function getAttributes() {
            if ($this->attributes == NULL)
                $this->attributes = new EstadoAtributos($this);

            return $this->attributes;
        }

        function getViews() {
            if ($this->views == NULL)
                $this->views = new EstadoVisoes($this);

            return $this->views;
        }

        function getRelations() {
            if ($this->relations == NULL)
                $this->relations = new EstadoRelacionamentos($this);

            return $this->relations;
        }

    }

?>
```

ANEXO C – Exemplos de interfaces PCL (*PHP Custom Library*) geradas

Interface principal



Interface de lista



Interface de manutenção