GP - Uma ferramenta de geração iterativa de código fonte utilizando o Método Rápido de Desenvolvimento de Sistemas (MRDS)

Márcio Alexandre de Oliveira Pinto

Instituto de Matemática - IM
Núcleo de Computação Eletrônica - NCE
Universidade Federal do Rio de Janeiro – UFRJ
Mestre em Informática

Prof. Eber Assis Schmitz, Ph.D.
NCE/UFRJ

Rio de Janeiro, RJ – Brasil
Março - 2004

GP - Uma ferramenta de geração iterativa de código fonte utilizando o Método Rápido de Desenvolvimento de Sistemas (MRDS)

Márcio Alexandre de Oliveira Pinto

Tese submetida ao corpo docente do Instituto de Matemática / Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro – UFRJ - como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada por:

_____

Prof. Eber Assis Schmitz, Ph.D.
DCC-IM/NCE-UFRJ (Orientador)

_____

Prof. Carlo Emanuel Tolla de Oliveira, Ph.D.
DCC-IM/NCE-UFRJ

_____

Prof. Toacy Cavalcante de Oliveira, D.Sc.
PUC-Rio

Rio de Janeiro, RJ – Brasil

Março - 2004

Pinto, Márcio Alexandre de Oliveira.

GP - Uma ferramenta de geração iterativa de código fonte utilizando o Método Rápido de Desenvolvimento de Sistemas (MRDS). Rio de Janeiro: UFRJ/IM/NCE, 2004

Dissertação (Mestrado) - Universidade Federal do Rio de Janeiro / Instituto de Matemática / Núcleo de Computação Eletrônica, Rio de Janeiro, 2004. Orientador: Eber Assis Schmitz.

1-Geração de Código 2-UML 3-OO 4-Arquitetura de Software 5-FAST CASE I. Universidade Federal do Rio de Janeiro II. Eber Assis Schmitz III. Titulo

Aos meus pais.

Agradecimentos

Primeiramente a Deus por ser fonte inesgotável de luz que ilumina e guia todos os meus passos, e que sem Ele nada é possível.

Ao professor Eber Schmitz, pelo apoio e constante confiança na conclusão deste projeto.

Aos meus pais e irmãos pelo ininterrupto suporte moral ao longo destes anos de Mestrado.

À Cristiane Opalka, pelo apoio e suporte, e por compreender e permitir que o tempo que seria dedicado e ela fosse dividido com o projeto.

Aos colegas do grupo FAST CASE do professor Eber Schmitz, em especial aos professores Denis Silveira e Pedro Oscar pelas opiniões e boas idéias.

Ao professor Denis Silveira, também pela sua inestimável colaboração na revisão deste material.

Aos diversos alunos da disciplina de Modelagem de sistemas de informação II da graduação do departamento de ciência da computação da UFRJ, que tiveram paciência e contribuíram com idéias e sugestões no aperfeiçoamento do trabalho.

A D. Deise e sua equipe de secretaria, em especial à Regina, por desatarem os mais tenebrosos nós burocráticos.

Ao SERPRO, em especial ao meu chefe Alfredo que permitiram e deram o apoio necessário para a conclusão do trabalho.

RESUMO

PINTO, Márcio Alexandre de Oliveira. **GP - Uma ferramenta de geração iterativa de código fonte utilizando o Método Rápido de Desenvolvimento de Sistemas (MRDS);** Orientador: Eber Assis Schmitz. Rio de Janeiro: UFRJ/NCE; Março/2004. Dissertação (Mestrado em Informática).

Este trabalho apresenta uma ferramenta interativa para suporte ao desenvolvimento de software que implementa um módulo de geração de código fonte na arquitetura para a "Metodologia Rápida de Desenvolvimento de Sistemas" (MRDS) e outro módulo de recuperação de código fonte permitindo uma nova geração sem perda de informações. Esses módulos permitem a possibilidade de um desenvolvimento incremental de sistemas, a ferramenta é baseada em um subconjunto da UML e utiliza a linguagem Object Pascal como destino de geração de código e como origem dos dados de recuperação. Trabalha para sistemas do domínio de aplicações comerciais "transacionais".

A ferramenta desenvolvida tem como objetivo ser uma ferramenta para uso em ambiente de ensino, proporcionando aos alunos uma maneira eficiente de aprendizado de arquitetura de sistemas permitindo um desenvolvimento interativo, incremental e produtivo, sem a perda das informações. Está acoplada à ferramenta FAST CASE, que é uma ferramenta CASE para suporte ao desenvolvimento de sistemas orientado a objetos, também voltada para o ambiente de ensino.

ABSTRACT

PINTO, Márcio Alexandre de Oliveira. **GP - Uma ferramenta de geração iterativa de código fonte utilizando o Método Rápido de Desenvolvimento de Sistemas (MRDS);** Orientador: Eber Assis Schmitz. Rio de Janeiro: UFRJ/NCE; Março/2004. Dissertação (Mestrado em Informática).

This work presents an interactive tool to support the software development that implements a module of code generation in the architecture for the "Metodologia Rápida de Desenvolvimento de Sistemas" (MRDS) and another module of code recovery allowing a new generation without loss of information. These modules allow the possibility of a incremental development of systems, the tool is based on a subgroup of the UML and uses Object Pascal language as destination of code generation and as origin of the recovery data.

The developed tool has as objective to be a tool for use in environment of education, providing to the students an efficient way of learning of architecture of systems allowing an interactive, incremental and productive development, without the loss them information. It is connected to tool FAST CASE, that it is a CASE tool for support to the development of systems oriented of objects, also to the education environment.

1 Introdução 17

1.1 Apresentação 17

1.2 Motivação e objetivos do trabalho 20

1.3 Estrutura do trabalho 21

2 Revisão de literatura 22

2.1 Geradores de artefatos 22

2.2 A Geração automática de Código 24

2.2.1 Estratégias de geração de código 24

2.2.1.1 Estratégias Parciais 25

2.2.1.2 Estratégias Generativas 26

2.2.1.3 Estratégias Transformacionais Completas 27

2.2.1.3.1 A Estratégia do Modelo Refinado 28

2.2.1.3.2 A Estratégia do Compilador Refinado 29

2.3 MDA e UML executável (xUML) 30

2.4 Arquitetura de Software 33

2.4.1 A Importância da Arquitetura 34

2.5 O Padrão MVC 37

2.5.1 Vantagens do MVC 39

2.5.2 Struts 40

2.6 O Mapeamento de Objetos para Banco de Dados Relacional 41

2.6.1 A importância dos Identificadores de Objetos.....42

2.6.2 Mapeando Atributos para Colunas.....42

2.6.3 Mapeando Classes para Tabelas.....43

2.6.3.1 Implementando Herança no Banco Relacional.....43

2.6.3.2 Uma Entidade para Todas as Classes de uma Hierarquia.....43

2.6.3.3 Uma Entidade para Cada Classe Concreta.....44

2.6.3.4 Uma Entidade por Classe.....45

2.6.4 Mapeando Associações, Agregação e Composição.....47

2.6.5 A diferença entre associação, agregação e composição.....47

2.6.6 Implementando relacionamentos em banco relacional.....48

2.6.7 Implementando associações muitos-para-muitos.....49

2.7 A Metodologia Rápida de Desenvolvimento de Sistemas.....50

2.7.1 Modelo de Requisistos.....52

2.7.2 Modelo de Análise e Projeto.....54

2.8 Os Diagramas da UML Utilizados.....56

2.8.1 Diagrama de Casos de Uso.....57

2.8.2 Diagrama de Classes.....59

2.8.3 Diagrama de Seqüência.....60

2.8.4 Diagrama de Transição de Estados.....62

2.9 O FAST CASE.....63

3 O Gerador de Programas.....65

3.1 A arquitetura esperada. 65

3.2 O Mapeamento dos modelos UML 68

3.2.1 Os Casos de Uso do tipo manutenção 69

3.2.2 Armazenamento de dados: Classes do domínio 69

3.2.3 Interface com o usuário: Classes de interface....74

3.2.4 Fluxo de controle do sistema: Classes de controle 75

3.2.5 A recuperação das informações....77

3.3 Processo de desenvolvimento usando o Gerador de Programas 78

3.3.1 Preparar Modelo no Fast Case 79

3.3.2 Disparar GP 81

3.3.3 Recuperar Informações 82

3.3.4 Definir Parâmetros no Gerador de Programas....84

3.3.5 Gerar Código 87

3.3.6 Complementar Código 89

3.4 Detalhamento interno do Gerador de Programas....90

3.4.1 Disparar o GP....91

3.4.2 A atividade Gerar Código 92

3.4.2.1 Ordenar Classes de domínio....93

3.4.2.2 Gerar Arquivos Padrões 94

3.4.2.3 Gerar as classes de domínio 96

3.4.2.4 Gerar as classes de interface....97

3.4.2.5 Gerar as classes de controle.....99

3.4.3 A atividade Recuperar Informações.....100

4 Avaliação da ferramenta.....101

4.1 Evolução da ferramenta.....101

4.2 Estudos de casos.....103

4.3 Um exemplo com a versão atual.....103

4.3.1 Primeira iteração.....104

4.3.2 Segunda iteração.....106

4.3.3 Terceira iteração.....108

5 Críticas e Comparações.....115

6 Conclusões.....118

7 Sugestões de trabalhos futuros.....120

8 Apêndices.....121

8.1 Apêndice A.....121

8.1.1 Ordenação Topológica.....121

8.2 Apêndice B.....122

8.3 Apêndice C.....137

8.3.1 Tipos Procedurais.....137

9 Referências Bibliográficas.....141

Lista de Figuras

Figura 1-1 O desenvolvimento utilizando o Gerador de Programas. 19

Figura 2-1 Estratégias de geração de programa 25

Figura 2-2 Arquitetura MVC segundo (SINGH, 2002). 38

Figura 2-3 Diagrama de classe de uma hierarquia simples 43

Figura 2-4 Uma tabela para toda hierarquia.....44

Figura 2-5 Uma entidade de dados para cada classe concreta.....45

Figura 2-6 Uma entidade de dados por classe.....46

Figura 2-7 Diferença entre associação e agregação/composição 48

Figura 2-8 Implementando relacionamentos (AMBLER, 2000b).....49

Figura 2-9 Relacionamento muitos-para-muitos.....49

Figura 2-10 Implementação de relacionamento muitos-para-muitos. 50

Figura 2-11 Diagrama esquemático da MRDS. 52

Figura 2-12 Meta-modelo do Modelo de Requisitos. 53

Figura 2-13 Meta-modelo do Modelo de Análise e Projeto. 55

Figura 2-14 Particionamento das funcionalidades do Caso de Uso 58

Figura 2-15 Instâncias das respectivas classes (PINTO e BASTOS, 2000) 59

Figura 2-16 Estrutura centralizada e estrutura descentralizada 62

Figura 2-17 O Fast Case. 64

Figura 3-1 Interface principal do Gerador de Programas. 65

Figura 3-2 As classes da arquitetura padrão (SILVEIRA e SCHMITZ, 2001) 66

Figura 3-3 Visão global do mapeamento feito pelo GP. 68

Figura 3-4 Forma Canônica para Mapeamento de Classes de Domínio (SCHMITZ e SILVEIRA, 2000).....70

Figura 3-5 Estrutura básica do domínio(PINTO e BASTOS, 2000) 71

Figura 3-6 Constantes com os nomes dos arquivos paradox.....71

Figura 3-7 Classes Básicas do domínio. 72

Figura 3-8 Exemplo de Classes Tgerente e TumObjeto.....73

Figura 3-9 Exemplo de código da classe Tgerente. 73

Figura 3-10 Função e procedimento para acesso a propriedades. 74

Figura 3-11 Herança entre Classes de Interface.....75

Figura 3-12 Estrutura do método TrataEvento.....76

Figura 3-13 DTE de Caso de Uso do Tipo Mantendo.....77

Figura 3-14 Estrtura do arquivo de projeto do Delphi. 77

Figura 3-15 Processo da utilização do GP 78

Figura 3-16 Detalhamento do processo Implementar Versão do SI. 79

Figura 3-17 Tela onde é feito o relacionamento entre os casos de uso e o diagrama de seqüência. 81

Figura 3-18 Tela onde éinformado os casos de uso do tipo mantendo. 82

Figura 3-19 Menu recupera informações. 83

Figura 3-20 Código adicionado manualmente e recuperado para nova geração. 83

Figura 3-21 Menu assistente de interface. 84

Figura 3-22 O assistente de criação de interface (GUI). 85

Figura 3-23 Opções de Units. 86

Figura 3-24 Opções de domínio. 86

Figura 3-25 Visualização do Script de criação do banco de dados. 87

Figura 3-26 Visualização do código que será gerado. 88

Figura 3-27 Processos internos do GP. 91

Figura 3-28 Modelo de classes de domínio do GP. 92

Figura 3-29 Sub-atividades da atividade Gerar código. 93

Figura 3-30 Gerar arquivos padrões 94

Figura 3-31 Gerar classes domínio 96

Figura 3-32 Método Create do DataModule. 96

Figura 3-33 Gerar classes de interface 97

Figura 3-34 Gerar classes de controle 99

Figura 3-35 Sub-atividades da atividade Recuperar Informações 100

Figura 4-1 Casos de uso da primeira iteração. 104

Figura 4-2 Diagrama de transição de estados da classe de controle do caso gerenciando paciente. 105

Figura 4-3 O GP pronto para a primeira geração. 106

Figura 4-4 Casos de uso após a segunda iteração. 107

Figura 4-5 O GP pronto para a segunda geração. 108

Figura 4-6 O diagrama de classes do domínio. 109

Figura 4-7 Maquete importada para o GP e regerada. 110

Figura 8-1 Invocação de método por referência de classe. 138

Figura 8-2 Tipos procedurais.....138

Figura 8-3 Declarações de vários tipos procedurais.....139

Figura 8-4 Ponteiros para métodos 140

Figura 8-5 Wrapper para rotinas predefinidas 140

Lista de Tabelas

Tabela 2-1 Comparando as estratégias de mapeamento de herança. 47

Tabela 3-1 Estrutura das pastas e arquivos gerados. 89

Tabela 4-1 Resumo do histórico das versões do GP 102

Tabela 4-2 Relação entre os modelos de cada iteração e suas saídas. 111

Tabela 4-3 Relação entre as iterações e a quantidade de classes.....112

Tabela 4-4 Relação dos números de linhas geradas.....113

1 Introdução

1.1 Apresentação

A construção de *software* cada vez mais precisa de ferramentas que automatizem seu processo. As várias conferências dedicadas ao tópico de Ambientes de apoio ao desenvolvimento ou sistemas CASE mostram o interesse neste tópico pela comunidade de software. Também são vários os fornecedores comerciais de produtos que apoiam o desenvolvimento, e alguns desses produtos oferecem também algum suporte na geração de código. Cada um deles tem vantagens e desvantagens e a escolha da ferramenta mais adequada não é uma tarefa fácil. (PINTO, 2000)

Este trabalho apresenta o resultado do desenvolvimento de uma ferramenta interativa, chamada de Gerador de Programas (GP), para suporte ao desenvolvimento de software e é composta de dois módulos: um módulo de geração de código fonte seguindo uma arquitetura que se baseia no padrão Modelo – Visão – Controle (MVC), descrita em BURBECK, e um segundo módulo que permite a recuperação de código fonte gerado pelo módulo citado anteriormente permitindo uma geração posterior sem a perda de informação já adicionada ao sistema manualmente[1]. Ao utilizar esses dois módulos, o projetista pode, naturalmente, desenvolver sistemas de uma forma incremental.

O trabalho apresenta além do GP, um novo processo para o desenvolvimento de sistemas, baseada na Metodologia Rápida de Desenvolvimento de Sistemas (MRDS), descrita em SCHMITZ e SILVEIRA (1999) e SILVEIRA (1999). Este novo processo incorpora o conceito de round-trip a MRDS. A ferramenta desenvolvida está acoplada

[1] Processo conhecido como “round-trip engineering”.

ao FAST CASE[2], ferramenta desenvolvida e apresentada por SILVEIRA (1999). O FAST CASE é uma ferramenta CASE que dá suporte a MRDS.

A característica da engenharia “round-trip” é a repetição cíclica do processo de geração do código fonte a partir de um modelo, o refinamento do código fonte e testes de avaliação da versão seguidos de uma atualização do modelo através do processo de recuperação destas informações do código fonte. Esse processo permite a geração rápida de novas versões do programa fonte mantendo, ao mesmo tempo, a compatibilidade do modelo.

O GP utiliza um subconjunto da UML como documentos de entrada produzindo como saida um programa fonte na linguagem Object Pascal. No passo reverso, programas fonte em Object Pascal são a origem para atualização do modelo em UML.

O GP foi desenvolvido para ser utilizada em cursos de graduação em computação, proporcionando aos alunos uma maneira prática de aprendizado de métodos de desenvolvimento de software. Um dos maiores problemas no ensino de métodos de desenvolvimento de sistemas encontra-se nos trabalhos práticos. Normalmente, os trabalhos práticos das disciplinas de graduação de informática limitam-se a exercícios de construção de modelos, impedindo que o aluno consiga fazer uma ponte entre os conceitos do modelo e sua implementação. Uma ferramenta de geração de código fonte permite que desde as primeiras aulas da disciplina, os alunos possam ir gerando programas que acompanham as evoluções do modelo.

Mais especificamente, o processo de desenvolvimento usando o GP consiste de duas atifvidades principais. A primeira atividade recebe como entrada modelos da UML de um sistema (*estáticos e dinâmicos*), através da leitura do repositório do FAST CASE, e produz como saída o esqueleto do código fonte para este sistema no ambiente *Delphi* e seu mapeamento em uma base de dados (através de “scripts” ou a própria base). A

[2] O FAST CASE é uma ferramenta CASE que dá suporte a MRDS.

segunda atividade consiste em receber como entrada o código fonte em *Delphi*, gerado em uma iteração anterior do processo, e recuperar as informações adicionadas ao código, especificamente à implementação dos métodos, para que seja possível uma nova iteração sem a perda de informações já adicionadas manualmente ao código e que não são representadas nos modelos.

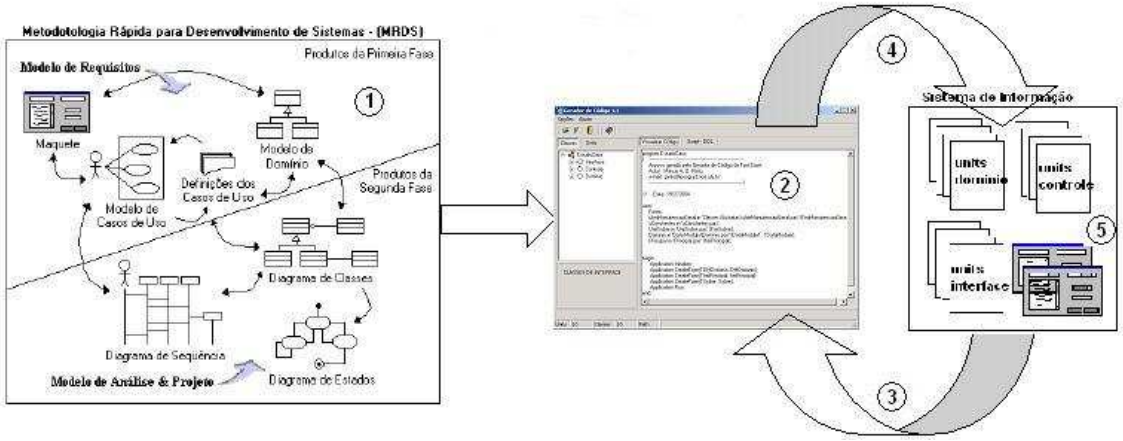

Figura 1-1 O desenvolvimento utilizando o Gerador de Programas.

A figura 1-1 mostra, de forma simplificada, uma visão macro do processo de desenvolvimento de um sistema de informação utilizando o GP. No passo “1” são feitos os modelos do sistema no FAST CASE, em seguida, passo “2” o GP é acionado e algumas customizações são possíveis de serem feitas na ferramenta, em seguida temos o passo “3”, onde ocorre a recuperação das informações completadas manualmente em código que foi gerado anteriormente, observe que esse passo não acontece na primeira iteração do processo, pois não temos o que recuperar e consequentemente não deve ser executado, o passo “4” é o acionamento da geração de código propriamente dita e que irá gerar os arquivos e artefatos do sistema de informação, o passo “5” é onde podemos completar o corpo dos métodos das classes nos artefatos gerados, esse é o último passo de cada iteração, a partir daqui devemos então retornar ao passo “1” e repetir todo o processo. Podemos observar que temos como entradas para a ferramenta, ou os modelos da MRDS, ou os arquivos gerados pelo próprio GP, e como saída os arquivos contendo a codificação dos sistema de informação.

Não é objetivo do GP aqui apresentado, gerar completamente o código fonte, nem recuperar totalmente a informação do código fonte. Além disso, não é propósito do GP gerar de forma automática a implementação dos métodos das classes, e sim eles serem incluídos manualmente pelo aluno. No processo de recuperação de informação, o GP recupera essas implementações dos serviços criando uma disciplina no seu uso. O GP também não verifica possíveis inconsistências do modelo de dados, apesar de realizar algumas verificações necessárias para o funcionamento correto, tais como a existência de ciclos de herança entre as classes persistentes. O GP também não oferece qualquer auxílio na geração de casos de teste.

1.2 Motivação e objetivos do trabalho

Existiram alguns fatores que motivaram a implementação e desenvolvimento da ferramenta e do processo aqui apresentados. Como já foi citado anteriormente, o ensino de uma metodologia para desenvolvimento de sistemas requer que seja realizada a implementação de um sistema para melhor aproveitamento e entendimento do que foi apresentado teoricamente. Porém uma implementação sem o auxílio de uma ferramenta demanda muito tempo, e os cursos de graduação possuem tempo limitado, o que não permitia que o aluno conseguisse implementar um sistema, respeitando uma arquitetura específica, que está sendo absorvida e é uma novidade para o aluno, e utilizando uma metodologia que também está sendo colocada em prática pela primeira vez.

O GP vem auxiliar ao aluno a ter uma visão ampla de arquitetura de software e da metodologia adotada, permitindo um ganho de tempo no aprendizado dos conceitos sobre a arquitetura de software e da metodologia proposta possibilitando ao mesmo implementar um sistema dentro das características esperadas em um período mínimo de tempo, já que permite um aumento de produtividade em ações mecânicas e repetitivas.

O objetivo deste trabalho foi produzir uma ferramenta simples onde o aluno não gaste muito tempo em aprender o funcionamento desta e não precise de auxílio extra para poder utilizá-la, que seja de uso público e de fácil instalação, evitando problemas de incompatibilidades e transporte da ferramenta para outras máquinas em que o usuário

deseje trabalhar. Com isso, o aluno concentra-se muito mais em assimilar os conceitos de modelagem e arquitetura de software podendo ter um exemplo prático em um sistema desenvolvido e idealizado pelo próprio aluno, facilitando a absorção dos conceitos apresentados teoricamente.

A ferramenta tem como escopo a geração e apoio ao desenvolviento de sistemas de informação monousuário.

1.3 Estrutura do trabalho

O capítulo 2 deste trabalho descreve os métodos e técnicas envolvidas na geraçao de artefatos e geração automática de programas, temos uma breve introdução a MDA e a UML executável, é apresentado também uma seção sobre arquitetura de software, o padrão de arquitetura MVC, descreve o problema do mapeamento entre os paradigmas OO e relacional,, descreve a Metodologia Rápida de Desenvolvimento de Sistemas, apresenta os diagramas da UML utilizados pelo GP e o FAST CASE. O capítulo 3 descreve a concepção e a implementação da ferramenta de geração de programas. É apresentado tanto o ponto de vista externo, demonstrando o processo de utilização do GP, como do ponto de vista interno, mostrando o comportamento interno do GP. O capítulo 4 apresenta à avaliação da ferramenta baseada em diversos experimentos de seu uso, alguns deles em cursos realizados no Departamento de Ciência da Computação da UFRJ. O capítulo 5 apresenta as críticas e comparações com trabalhos existentes e relacionados ao projeto aqui apresentado. O capítulo 6 a conclusão, o capítulo 7 as considerações para trabalhos futuros o 8 contém os apêndices e no 9 as referências bibliográficas utilizadas.

2 Revisão de literatura

2.1 Geradores de artefatos

Os geradores de artefatos são softwares que produzem um artefato a partir de sua especificação de alto nível. Os artefatos gerados podem ser: programas completos, módulos, documentação, arquivos de configuração, plano de teste, etc. (FRANCA, 2000)

A utilização de geradores de artefatos não é uma idéia nova (FRANCA, 2000). Em JENKINS (1985) já foi apresentado um *survey* dos geradores disponíveis comercialmente naquela época. Para CLEAVELAND (1988), MASIERO e MEIRA (1993) e NEIGHBORS, BIGGERSTAFF e PERLIS (1989), a maioria dos geradores analisam uma especificação para então fazer a geração do artefato.

Um processo de desenvolvimento baseado em geradores preconiza que a menutenção do artefato gerado seja sempre e integralmente realizada na sua especificação e não nos seus arquivos de implementação. (FRANCA, 2000)

Quando uma organização decide utilizar geradores de artefatos dentro dos seus processos de desenvolvimento, ela tem como principal objetivo a promoção do reuso de largo espectro (PFLEEGER, 1998). Diferentes relatos apresentados na literatura, como em BASSET (1997), mostram resultados similares bastante expressivos em relação à adoção de geradores.

Segundo FRANCA (2000), os principais benefícios esperados com a utilização de geradores são: aumento de produtividade, redução de tempo para o mercado, prototipação e qualidade do artefato gerado.

Uma boa fonte de referências para construção de geradores de artefatos, é a literatura sobre *frameworks* (FRANCA, 2000). Os conceitos e as experiências no desenvolvimento de *frameworks* podem ser aplicados dentro do contexto de geradores

de artefatos.

Foi observado em FRANCA (2000) que o papel de ambiente de geração pode ser exercido por um CASE que tenha facilidades de customização. Em HOHENSTEIN (2000) e MILICEV (2000) são descritos trabalhos nos quais ferramentas CASE são utilizadas como ambiente de geração. Em HOHENSTEIN (2000) é apresentada uma lista das caracetrísticas que uma ferramenta CASE deve ter para ser utilizada como ambiente de geração:

- Possuir um repositório contendo todos os dados dos modelos que estão sendo utilizados;
- Disponibilizar interfaces que permitam amplo acesso à meta-dados do repositório;
- Possuir mecanismos para editar os meta-modelos disponíveis, adicionando ou modificando seus meta-dados;
- Permitir a criação e a execução de scripts que exploram e manipulam o conteúdo do repositório.

No trabalho apresentado por FRANCA (2000), é adotado o CASE Talisman (STAA, 1993) como gerador de geradores de artefatos, justificada pela existência das características desejáveis para um ambiente de geração.

Em FRANCA (2000), é definido um processo de construção de geradores de artefatos. O ponto de partida do processo proposto é o artefato-exemplo, tendo a necessidade de que esse artefato-exemplo possua uma elevada qualidade, uma vez que o gerador replicará a sua implementação. O resultado do processo de contrução proposto em FRANCA (2000) é um gerador capaz de gerar artefatos similares ao artefato-exemplo.

2.2 A Geração automática de Código

A idéia central atrás da geração de código, por SELIC (2000), é a tradução de um modelo abstrato formal de alto nível em um programa de computador, com funcionalidade equivalente, escrito em uma línguagem de programação padrão (por exemplo: Pascal, C, C++, Java, etc.).

Isto levanta as seguintes perguntas: porque precisamos primeiro de um modelo? Por que não ir direto para codificação? A razão fundamental para a construção de modelos é a necessidade de gerenciar a complexidade dos sistemas a um custo razoável. No caso particular dos sistemas de informação, a complexidade ocorre devido a inúmeros fatores: desde a quantidade e sofisticação dos requisitos até a complexidade da tecnologia de implementação (SELIC, 2000).

2.2.1 Estratégias de geração de código

SELIC (2000), ressalta três abordagens para a geração de código. A saber:

- A abordagem mais simples para geração automática de código é gerar uma estrutura genérica e deixar a tarefa de preencher os detalhes que faltam e resolver a ambigüidade para os desenvolvedores. Isto é, somente a parte das chamadas para a implementação é gerada automaticamente a partir do modelo, enquanto o resto é preenchido através de código escrito manualmente. Por razões óbvias, essa abordagem é chamada de estratégia parcial de geração de código.

- Uma abordagem diferente é a representada por estratégias generativas. Estas são estratégias em que o gerador de código sintetiza automaticamente o detalhe que falta e resolve a ambigüidade usando alguma política padrão de construção.

- A terceira abordagem é representada pelas estratégias transformacionais. Neste caso, o detalhe que falta é especificado diretamente pelo

desenvolvedor, mas, em contraste com os métodos parciais, este é ligado formalmente com o modelo. A ligação formal garante que o detalhe e o modelo estão semanticamente sincronizados, fazendo com que esse método seja mais confiável que as estratégias parciais. Existem duas maneiras diferentes em que isto é feito: (1) unindo o detalhe à parte correspondente no modelo e (2) fornecendo ao compilador do modelo as regras formais que controlam a geração do detalhe que falta.

A figura 2-1 mostra a taxonomia das estratégias de geração de código fonte, segundo SELIC (2000), que serão descritas com mais detalhes nas próximas seções.

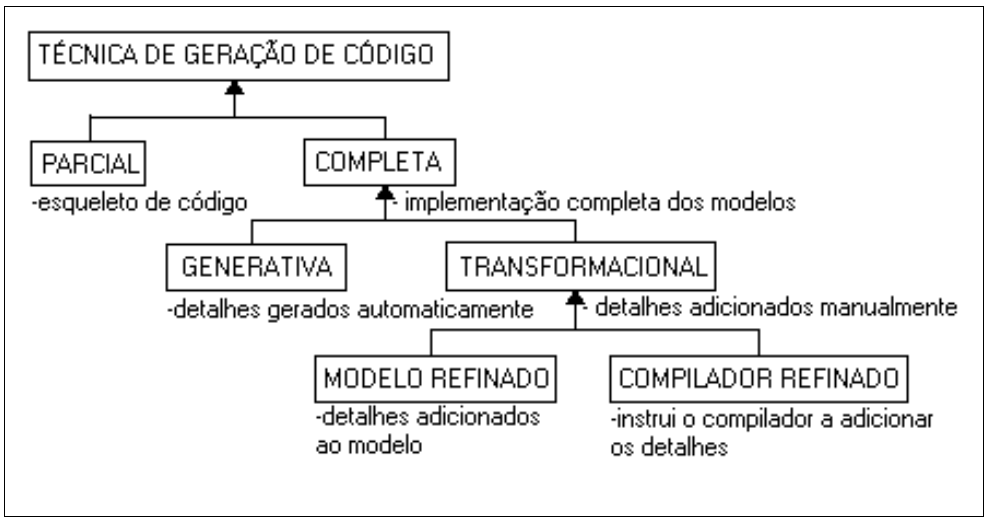

Figura 2-1 Estratégias de geração de programa

2.2.1.1 Estratégias Parciais

De acordo com SELIC (2000), as estratégias parciais são também conhecidas como técnicas de geração de esqueleto de código, já que geram uma estrutura básica do código (esqueleto) para que sejam adicionados o restante do código manualmente. O esqueleto gerado contém tipicamente a informação estrutural, embora alguns aspectos comportamentais básicos possam também ser gerados para abstrações comportamentais simples, tais como uma máquina não hierárquica de estados.

Segundo SELIC (2000), o benefício das estratégias parciais é limitado. Talvez, o

maior inconveniente é que não existe certeza que o modelo e o código implementado são mutuamente consistentes. Uma vez que o esqueleto foi gerado e modificado com a adição de código manualmente, o *link* semântico é quebrado. Não há nenhuma maneira de garantir que a implementação representa realmente o modelo, que, apesar de tudo, é supostamente o anteprojeto para a implementação.

Para contornar tais problemas, SELIC (2000) ressalta que algumas ferramentas CASEs fornecem a possibilidade da engenharia reversa. Esta é uma característica em que um corpo existente de código pode ser automaticamente capturado para o modelo (o código adicionado manualmente é então anexado, normalmente como comentário, na parte apropriada do modelo). Isto é muitas vezes chamado de "*round tripping*". Enquanto esta solução soa bem para resolver este problema, ela tem uma falha maior e fundamental: ela não serve para sistemas grandes.

Deve ser lembrado que o propósito de um modelo, segundo SELIC (2000), é abstrair os detalhes. Desta forma, uma vez que os detalhes são anexados, uma ferramenta de engenharia reversa automática completa não pode distinguir o detalhe das partes de alto nível do modelo. Em outras palavras, abstração é uma atividade unicamente humana, altamente baseada na intuição e nos processos que são difíceis, se não impossíveis de automatizar. Sem a possibilidade da abstração, a engenharia reversa de um modelo de um sistema complexo é quase tão difícil quanto navegar e compreender o próprio código fonte.

Alguma forma de engenharia reversa assistida pode ser uma solução para este problema. Todavia, tais ferramentas são ainda tópicos de pesquisas e suas utilidades ainda precisam ser provadas.

2.2.1.2 Estratégias Generativas

Em SELIC (2000), o autor explica que neste caso, os detalhes que faltam são gerados automaticamente. Claro, isto somente é aceitável se nós não temos um maior interesse com os detalhes que ainda faltam. Ou seja, tais aspectos como a definição precisa da estrutura de dados, desempenho do código gerado, a utilização da memória e

dos recursos do processador, devem ser aspectos irrelevantes ou de menor interesse.

2.2.1.3 Estratégias Transformacionais Completas

A característica que diferencia esta estratégia, das demais aqui apresentadas, é que a implementação completa é gerada, ou através do uso do modelo, que tem os detalhes que faltam embutidos, ou fornecendo ao compilador do modelo as informações apropriadas de como gerar as informações que faltam.

Em qualquer dos casos, segundo SELIC (2000), uma questão importante a ser respondida é se uma vez o código gerado automaticamente, pode ser modificado manualmente? Note que, quando o código é gerado a partir de um compilador tradicional de linguagem de alto nível, nós normalmente não modificamos o código objeto. Ao invés disso, se alguma coisa precisar ser corrigida, nós voltamos no código fonte e fazemos a nossa mudança lá. Por que devemos fazer diferente para uma linguagem de mais alto nível? Nós sabemos que o inconveniente da mudança manual da saída de um compilador inclui a possibilidade de inadvertidamente e transparentemente corromper o programa e a possibilidade de perda das mudanças manuais uma vez que o programa de alto nível é recompilado. O senso comum nos diz que, como regra, nós não devemos modificar o código gerado automaticamente. O mesmo problema acontece com linguagens de mais alto nível já que a complexidade da tradução automática é grande.

Contudo, é importante ressaltar a necessidade, em algumas circunstâncias, de modificar manualmente o código gerado automaticamente. Isto inclui:

- a ineficiência do código gerado;

a geração automática de código pode ter custo elevado conduzindo a tempos longos mesmo para mudanças localizadas e pequenas;

erros de compilação e de execução podem ser relatados no contexto da geração automática de código.

Com isso, SELIC (2000) conclui que para que uma geração de código automática completa seja útil e prática os seguintes itens têm que ser satisfeitos:

A geração de código deve ser tão eficiente quanto à codificação manual (este é o mesmo critério que foi usado por muitas das primeiras gerações dos compiladores de linguagens de alto nível (HLL) para assegurar a aceitação do mercado);

Deve ser possível embutir código manualmente dentro do modelo para áreas onde a geração de código automática é tida como inadequada.

O gerador de código deve ser altamente incremental com a propriedade de que uma alteração pequena e localizada pode ser re-gerada em um intervalo de tempo comparável á mudança equivalente para a linguagem de alto nível (HLL);

Todos os erros, em tempo de execução ou em tempo de compilação, devem ser relatados no modelo original e não no código gerado. Isso implica que um depurador de fonte de linguagem de mais alto nível (VHLL) deve estar disponível.

2.2.1.3.1 A Estratégia do Modelo Refinado

Nesta variante da técnica transformacional, o modelo de nível mais elevado é refinado acrescentado-se os detalhes de nível mais baixo necessários diretamente dentro do modelo – mais freqüentemente usando a linguagem alvo atual. Por exemplo, a ação tomada em uma transição de um estado para outro pode ser especificada em um pequeno fragmento de código C++. Este fragmento é uma parte integrante do modelo e é unida, tipicamente por *hiperlink*, à transição correspondente no modelo. O modelo completo é, segundo SELIC (2000), uma combinação da especificação do modelo em alto nível e suas ações especificadas detalhadamente unidas ao modelo. Quando a geração automática do código ocorre, os fragmentos explícitos do código são encaixados diretamente no código gerado, geralmente sem nenhuma modificação.

O valor deste tipo de estratégia diminui se a maioria do código tiver que ser realizado manualmente. Se este é ou não o caso, depende da riqueza semântica dos conceitos da modelagem de alto nível. Quanto mais rica for a semântica, maior o

benefício da geração automática de código. Com linguagens de modelagem mais sofisticadas, tais como a UML, a maioria da complexidade é contida no modelo e não no código de ação detalhado.

2.2.1.3.2 A Estratégia do Compilador Refinado

Essa variante especifica o detalhe independentemente do modelo. SELIC (2000) ressalta que ela proporciona um conjunto de diretivas para o compilador do modelo que descreve como os elementos do modelo são traduzidos. As diretivas são associadas ao modelo, mas eles são separados do modelo. Isto é porque um dos principais objetivos desta estratégia é permitir ao mesmo modelo ser flexível para implementação em diferentes ambientes.

Para SELIC (2000), esta é uma capacidade poderosa que permite que um modelo seja realmente independente da linguagem alvo. Por exemplo, é possível traduzir partes diferentes de um modelo em linguagens alvo diferentes sem ter que mudar o modelo. Entretanto, há alguns inconvenientes sérios para esta estratégia muito flexível. Os mais notáveis são:

Desde que ao menos alguma parcela das diretrizes de compilação é específica do modelo e pode mudar durante todo o desenvolvimento, é difícil assegurar o mesmo nível de exatidão do gerador de código que de um compilador estável. Isto é, o próprio compilador pode constantemente mudar. Quando os erros ocorrem, pode ser difícil localizar se o problema está no compilador do modelo ou no modelo. Muitos desenvolvedores do software são cientes da instabilidade introduzida na troca de compiladores ou mesmo versões do compilador.

A fragmentação do motor da geração do código do compilador do modelo faz com que fique muito difícil criar otimizações globais do tipo encontrado em otimizações em compiladores de linguagens de alto nível (HLL).

O forte desacoplamento entre os aspectos detalhados e o modelo de alto nível gera as descontinuidades que são às vezes difíceis de unir. Por exemplo, se o problema

for com as regras de tradução, eles não podem ser detectados através de um depurador de fonte de modelo de linguagem de mais alto nível (VHLL).

O tipo de habilidade requerido para especificar as diretrizes de um compilador é altamente especializada e radicalmente diferente da habilidade requerida para modelar. Esta habilidade é difícil e pode ter que ser retida por tanto tempo quanto o modelo está ativo.

Tais questões podem restringir um projeto fazendo esta técnica pouco prática. Como a tecnologia evolui, é possível que esta estratégia supere algum destes problemas e se torne mais prática. Entretanto, atualmente, são geralmente piores para uso industrial do que a técnica do modelo refinado.

2.3 MDA e UML executável (xUML)

A MDA (Model Driven Architecture) é padrão para especificação de sistema de TI que separa a especificação da funcionalidade da especificação da implementação dessa funcionalidade em uma plataforma de tecnologia específica. (OMG, 2003)

Em 2001, a OMG adotou o padrão MDA. A MDA é um padrão para o uso de modelos no desenvolvimento do software. A MDA é um passo pequeno na longa estrada a percorrer em uma disciplina de engenharia. A Arquitetura Dirigida a Modelo começa com a idéia estabelecida de separar a especificação da operação de um sistema dos detalhes da maneira que o sistema usa as potencialidades de sua plataforma. (OMG, 2003)

A MDA, segundo OMG (2003), fornece uma estratégia para, e permite que ferramentas sejam fornecidas para:

- Especificar um sistema independentemente da plataforma que o suporta
- Especificar plataformas
- Escolher uma plataforma particular para o sistema, e

- Transformar a especificação de sistema para uma plataforma particular.

Os três objetivos principais da MDA são portabilidade, interoperabilidade e reusabilidade com a separação arquitetural dos interesses. (OMG, 2003)

A MDA especifica três potos de vistas em um sistema, um ponto de vista independente de computação, um ponto de vista independente da plataforma e um ponto de vista específico da plataforma. (OMG, 2003)

O ponto de vista independente de computação, é composto pelo modelo independente de computação (CIM). Um CIM nao mostra os detalhes da estrutura do sistema. Um CIM é muitas vezes chamado de modelo de domínio e um vocabulário que seja familiar aos praticantes do domínio em questão é usado em sua especificação. (OMG, 2003)

O Modelo Independente de Plataforma (PIM) é a visão do sistema do ponto de vista independente de plataforma. O PIM exibe um grau específico de independência de plataforma para ser apropriado para o uso com um número de plataformas diferentes de tipos similares. (OMG, 2003)

Um Modelo Específico da Plataforma (PSM) é uma visão do sistema do ponto de vista específico da plataforma. Um PSM combina as especificações do PIM com os detalhes que especificam como esse sistema usa um tipo particular de plataforma. (OMG, 2003)

Segundo BROERING (2004) Model Driven Architecture (MDA) é um modo para separar a arquitetura de uma aplicação de sua implementação. Assim, seus proponentes esperam que a mudança de software e hardware não faça a aplicação empresarial tornar-se obsoleta antes do tempo. Mais importante, através do desaclopamento da arquitetura da aplicação de seu ambiente de execução, o uso da MDA pode resultar em melhores projetos para aplicações dando uma vida útil mais longa e portabilidade a outras plataformas.

MDA está baseado na Unified Modeling Language (UML), junto com Meta

Object Facility (MOF[3]) e Common Warehouse Metamodel (CWM[1]), além disso, a MDA tem alguns modelos de núcleo, ou perfis para desenvolvimento empresarial e outro para desenvolvimento em tempo real (tempo real no sentido de sistemas de hardware/software devem ter tempo de resposta previsível). Outros modelos de núcleo serão oferecidos como padrões com o passar do tempo. A MDA é uma especificação do Object Modeling Group (OMG), o mesmo grupo que mantém o CORBA e a UML. (BROERING, 2004)

Para BROERING (2004), o primeiro passo na aplicação MDA é criar o modelo de aplicação em UML, especificamente concentrando-se em áreas que definem sua arquitetura, comportamento, e colaboração. Há produtos disponíveis para geração de código a partir de diagramas UML, mas você está perdendo um ingrediente importante: as características do sistema operacional e middleware que a aplicação pode precisar usar. Exemplos são janelas e caixas de diálogo. Em muitos casos, eles tiram proveito de sistema operacional ou serviços da plataforma (no caso de Java) e é implementado com a plataforma específica.

O MDA captura o que um sistema faz em um modelo independente da plataforma (PIM), e captura como um sistema é projetado e implementado em um modelo específico da plataforma (PSM). O PIM pode ser completo e específico bastante para permitir a execução e o teste da aplicação, independentes de proejto e de implementações específicas. A automatização produtiva a partir de MDA fornece a oportunidade para a redução drástica de defeitos, o desenvolvimento acelerado, reuso em grande escala, e a migração fácil da plataforma das aplicações. (PROJECT, 2004)

Segundo MELLOR e BALCER (2002) a UML executável é a maior inovação na área de desenvolvimento do software. Deve ser usada para produzir um modelo detalhado e compreensível de uma solução independente da organização da implementação do software. É altamente abstrata que ajuda na formalização do

[3] Especificação da OMG (OMG, 2004).

conhecimento, e é também uma maneira de descrever os conceitos que tornam as soluções abstratas para os problemas do desenvolvimento do software.

Como o MDA, a UML executável fornece a tecnologia chave para expressar domínios da aplicação em uma maneira independente de plataforma. Mas a UML executável pode fazer mais do que formalizar os requisitos e casos de uso em um conjunto rico de diagramas de verificação. Os modelos têm uma ação semântica formal de modo que sejam executáveis e testáveis e podem ser traduzidos diretamente em código por compiladores de modelo executáveis de UML. (MELLOR e BALCER, 2002)

2.4 Arquitetura de Software

Encontramos na literatura algumas definições relacionadas ao termo arquitetura de software. Em GARLAN e SHAW (1993), os autores definem a arquitetura de um software como a descrição de elementos que construirão o sistema, interações entre estes elementos, padrões que guiarão suas composições e limitações destes padrões. Em geral, um sistema é definido em termos de uma coleção de componentes e interações entre estes componentes. Tal sistema pode, ainda, ser usado como um elemento de composição em um projeto de sistema maior.

Dentro do mesmo contexto, CLEMENTS (1996) define a arquitetura de software como a estrutura dos componentes do sistema, seus inter relacionamentos, princípios e diretrizes que governam seu projeto e evolução ao longo do tempo.

Em STAA (2000), o autor descreve que arquitetura é o processo de organização de um programa em termos dos módulos que os constituem e o seu resultado é a arquitetura do programa. O autor ressalta a idéia de que a arquitetura não deixa de ser uma forma de projeto, utilizada para estabelecer a organização global do programa.

Em GARLAN *et al* (1997), encontramos uma definição de arquitetura de software que preza cumprir dois papéis:

Fornecer um nível de abstração no qual os projetistas podem argumentar sobre o

comportamento do sistema: funcionalidade, desempenho, confiabilidade, e etc; e

Fornecer uma "consciência" para a evolução do sistema, indicando quais aspectos do sistema podem ser facilmente alterados sem comprometer a integridade do sistema.

A definição clássica apresentada por SHAW e GARLAN (1996) diz que arquitetura de software define o que é o sistema em termos de componentes computacionais e os relacionamentos entre estes componentes. Semelhante a esta definição BASS, CLEMENTS e KAZMAN (1998) diz que “arquiteturas de software são as estruturas que incluem componentes, suas propriedades externas e os relacionamentos entre eles, constituindo uma abstração do sistema. Esta abstração suprime detalhes de componentes que não afetam a forma como eles são usados ou como eles usam outros componentes, auxiliando o gerenciamento da complexidade”.

A arquitetura de software, de acordo com SILVEIRA e SCHMITZ (2001), estuda um conjunto de aspectos estruturais que envolvem a forma de organizar o sistema como um conjunto de componentes interconectados, as estruturas de controle responsáveis pela forma de sequenciamento do programa, os protocolos para comunicação, sincronismo e acesso a dados entre estes componentes, a alocação de funcionalidade a cada um dos componentes, a distribuição física destes componentes, estudos de desempenho e escalabilidade; e formas de evolução.

Segundo SILVEIRA e SCHMITZ (2001), o desenvolvimento da arquitetura de um software é análogo à criação da arquitetura de um prédio. Em ambos os casos, os arquitetos estão usando uma descrição simplificada para tornar possível a discussão de assuntos mais gerais, sem se preocupar, neste momento, com os problemas de construção dos componentes do sistema.

2.4.1 A Importância da Arquitetura

Um projeto arquitetural de sistemas possui um papel fundamental para o sucesso de todo o processo de desenvolvimento porque ele descreve o sistema através de um

nível mais alto de abstração, permitindo que sejam visualizadas as decisões referentes ao projeto o mais cedo possível, ajudando o desenvolvedor a verificar se os requisitos foram realmente atingindos. Através da abstração dos detalhes de implementação, uma boa descrição arquitetural facilita a gerência do projeto do sistema e expõe as propriedades mais cruciais para o seu sucesso.

Segundo VAROTO (2002) cada participante da construção do sistema está preocupado com características específicas que são afetadas pela arquitetura. Portanto, sua definição deve ser bem representada e documentada, usando uma notação a qual todos os participantes possam entender com facilidade.

Para BASS, CLEMENTS e KAZMAN (1998) isto não significa que o usuário final tenha que conhecer a notação na qual a arquitetura foi representada. Isto significa que a arquitetura do software representa uma abstração comum em alto nível de um sistema a qual a maioria dos participantes pode usar como base para criar um entendimento mútuo, formar consenso e comunicar-se uns com os outros.

As restrições e regras tanto do negócio quanto aquelas que influenciam os aspectos técnicos, segundo VAROTO (2002), devem ser atendidas pela arquitetura pois ela constitui-se de um modelo simples e inteligente de como o sistema deve ser estruturado e como seus componentes trabalham juntos.

Em BASS, CLEMENTS e KAZMAN (1998) verificamos também como um ponto possitivo o fato da arquitetura constitui também um ponto de referência comum para as demais atividades que são executadas posteriormente a sua definição. Isto significa que a arquitetura é a manifestação antecipada das decisões de projeto, preocupando-se com tempo de desenvolvimento, custo e manutenção, definição das restrições de implementação e definição da estrutura organizacional, enfatizando os atributos de qualidade que o sistema requer e medindo através de avaliações a empregabilidade das qualidades necessárias.

VAROTO (2002) ressalta que após extensas análises, avaliações e revisões da arquitetura, sua representação torna-se robusta o suficiente para guiar o projeto de

implementação, os testes e a implantação do sistema. Esta representação precisa estar congelada, evitando que as mudanças e decisões que podem ocorrer no meio das atividades de implementação coloquem em risco a construção do sistema.

Um dos objetivos da arquitetura, segundo JACOBSON *et al.* (1992), é promover a reutilização. Qualquer que seja o nível do reuso, é necessário que o sistema ou a aplicação possa sofrer alterações de forma localizada, sem afetar nenhuma outra parte, e que outras funcionalidades possam ser adicionadas sem impactos nas aplicações já existentes, com fácil interação. A vida útil do sistema que possui uma boa arquitetura é aumentada em função da facilidade na incorporação de mudanças.

Uma vez que a arquitetura pode ser definida como um conjunto de componentes computacionais (ou subsistemas) e o relacionamento entre eles, o reuso, segundo VAROTO (2002), é considerado uma característica muito forte no que diz respeito a tempo e custo de desenvolvimento do sistema. Se a arquitetura estiver bem organizada e estruturada, cada componente computacional ou parte dele pode ser construído com vistas para o reuso.

Ainda, segundo VAROTO (2002), podemos introduzir a definição de reuso considerando vários níveis de abstração:

1. **Reuso de idéias arquitetônicas**: consiste em uma das atividades de um bom arquiteto de software que se preocupa com a melhoria do seu trabalho e busca no registro de suas experiências as informações pertinentes para discernir sobre o que se adere bem e o que não se adere bem como solução técnica e de negócios ao problema em questão. Está relacionado com a gestão do conhecimento.

2. **Reuso de estilos e padrões de arquitetura**: consiste na utilização de estilos e padrões de arquitetura consagrados na literatura, adequando por semelhança o negócio à representação técnica disponível. Os estilos e os padrões fazem parte do kit do arquiteto como uma primeira referência para a escolha da arquitetura.

3. **Reuso de componentes de software**: consiste no reuso de partes codificadas

do sistema. Estas partes precisam encapsular funcionalidades e ter interfaces bem definidas de modo que possam ser facilmente aplicadas e substituídas. As partes podem ser desenvolvidas ou adquiridas.

O reuso é uma característica de qualidade agregada à definição da arquitetura. Portanto, de acordo com HOFMEISTER, NORD e SONI (2000), o reuso não deve ser obtido por acaso. Uma das motivações para o reuso é o fato dos desenvolvedores darem pouca importância à similaridade funcional. Por pressão de tempo de desenvolvimento, constroem sistemas poucos modularizados ou com módulos muito acoplados, dificultando a manutenção. (HOFMEISTER, NORD e SONI, 2000)

À medida que avança o desenvolvimento do sistema, VAROTO (2002) ressalta que novas partes vão sendo implementadas e conseqüentemente, a manutenção vai se tornando cada vez mais complexa. Aproveitar-se de estruturas ou componentes já prontas garante mais robustez para o sistema além de aumentar o grau de abstração, escondendo detalhes do problema que podem ser substituídos por uma solução pronta.

O emprego de reuso na arquitetura sugere uma implementação iterativa, com o mínimo necessário para que uma primeira versão esteja pronta no menor tempo possível. Assim, é possível avaliar a adequação e robustez da arquitetura em relação ao sistema, sem buscar de imediato a completude e perfeição desejada conforme os requisitos (VAROTO, 2002).

2.5 O Padrão MVC

O objetivo do padrão arquitetural MVC (*Model-View-Controller*) é propor a divisão da aplicação em três categorias pricipais de componentes. O componente “modelo” encapsula dados e funcionalidades pricipais da aplicação. O componente “visão” mostra informações ao usuário, obtidas de um modelo. Cada visão apresenta um componente “controlador” que recebe as entradas do usuário (na forma de eventos disparados) e os traduz em solicitações de serviços ao modelo e/ou visão.

Segundo BURBECK, no paradigma MVC a interação do usuário, a modelagem

do mundo externo, e o retorno visual para o usuário estão explicitamente separados e colocados em três tipos de objetos, cada um especializado em sua tarefa. A Visão gerencia a saída gráfica e/ou textual da porção que é mostrada da aplicação. O Controlador interpreta a interação do usuário feita pelo mouse ou teclado, comandando o Modelo e/ou a Visão realizar mudanças quando necessário. Finalmente, o Modelo gerencia o comportamento e os dados do domínio da aplicação, respondendo à solicitações sobre seu estado (geralmente vindo da Visão), e enviando instruções de mudança de estados (usualmente vindo do Controlador).

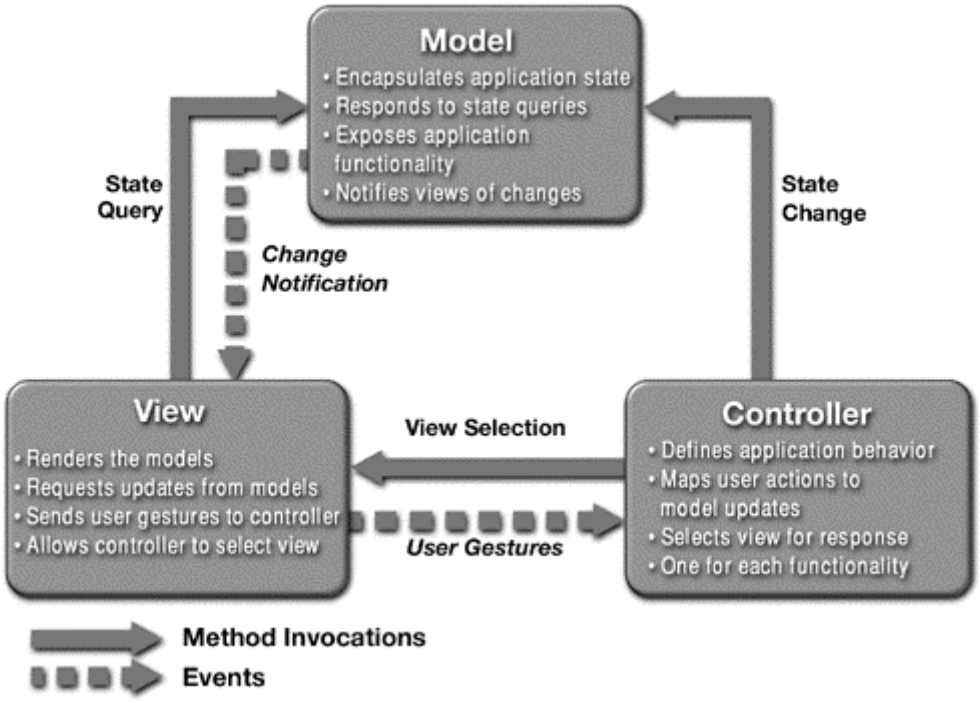

Figura 2-2 Arquitetura MVC segundo (SINGH, 2002).

O “modelo” sabe sobre todos os dados que precisam ser mostrados. Ele conhece também todas as operações que podem ser aplicadas para transformar os objetos. Todavia, ele não sabe nada sobre a interface gráfica com o usuário, a maneira com que os dados serão apresentados, nada sobre as ações da interface gráfica que serão usadas para manipular os dados. Os dados são acessados e manipulados através de métodos que

são independentes da interface gráfica. O “modelo” representa os dados e as regras de negócio que governam o acesso e a atualização desses dados. Frequentemente o “modelo” serve como uma aproximação para o processo do mundo real, então técnicas simples de modelagem do mundo real são aplicadas quando definimos o “modelo” (DASS).

A “visão” referencia o modelo. Ele usa os métodos de consulta do “modelo” para obter dados a partir do “modelo” e então mostrar a informação. A “visão” reproduz o conteúdo do “modelo”. Ela acessa os dados através do “modelo” e especifica como que os dados poderão ser apresentados. Isto é a responsabilidade da “visão” em manter a consistência na apresentação quando existe mudança no “modelo” (DASS).

O “controlador” conhece sobre os meios físicos pelo qual os usuários manipulam os dados com o “modelo”. O “controlador” traduz as interações com a “visão” em ações a serem executadas no “modelo”. Em uma interface gráfica cliente “*stand-alone*”, a interação do usuário pode ser um clique no botão ou uma seleção em um menu, onde em uma aplicação Web, elas aparecem como requisições HTTP de GET e POST. As ações realizadas pelo modelo incluem ativação de processos de negócio ou mudança de estado do “modelo”. Baseado nas interações dos usuários e nas saídas das ações do “modelo”, o “controlador” responde através da seleção a visão apropriada (DASS).

Nas interfaces gráficas, “visões” e “controladores” frequentemente trabalham muito juntos. Por exemplo, um “controlador” é reponsável por atualizar um parâmetro particular no “modelo” que é então mostrado pela “visão”. Em alguns casos um simples objeto pode funcionar tanto como “controlador” quanto como “visão”. Cada par “controlador” e “visão” é associado com somente um “modelo”, todavia um “modelo” particular pode ter muitos pares “controlador” e “visão” (DASS).

2.5.1 Vantagens do MVC

Em DASS são ressaltados os seguintes benefícios para a arquitetura MVC:

Múltiplas visões usando o mesmo modelo: A separação entre o Modelo e Visão

permite múltiplas visões para usar o mesmo modelo. Consequentemente, componentes do modelo de aplicação são fáceis para implementar, testar e manter, já que todos os acessos ao modelo são feitos através desses componentes.

Fácil suporte para novos tipos de clientes: Para suportar um novo tipo de cliente, você simplesmente escreve uma Visão e um Controle para ele e os conecta no modelo existente.

Claridade de design: apenas olhando a lista de métodos públicos do modelo, pode facilmente entender como o controle do modelo se comporta. Quando projetamos a aplicação, essa característica faz com que o programa inteiro seja fácil de implementar e manter.

Modularidade eficiente: do projeto permite que qualquer dos componentes possa ser trocado como o ususário ou o programador desejar. Mudanças em um aspecto do programa não está acoplado a outros aspectos, eliminando muitas situações de depuração desagradáveis. Permite também, o desenvolvimento de vários componentes que podem progredir em paralelo, uma vez que a interface entre os componentes é claramente definida.

Fácil crescimento: Controles e Visões podem crescer como o Modelo cresce; e versões de visões e controles antigos podem ainda ser usados contanto que uma interface comum seja mantida.

Distribuível: Com um par de proxies um pode facilmente distribuir qualquer aplicação MVC por alteração apenas no método de inicialização da aplicação.

2.5.2 Struts

O Struts, segundo (DAVIS, 2001), é uma implementação do padrão MVC (Model-View-Controller) que usa tecnologia servlets and JavaServer Pages (JSP). O Struts é uma implementação “open-source” do padrão MVC que facilita o desenvolvimento (facilitando a gerência da complexidade) em grandes web sites.

O Struts resolve alguns grandes problemas usando tags e MVC. Esta aproximação ajuda no reuso de código e na flexibilidade. Separando o problema em componentes menores você estará mais apto ao reuso quando as mudanças ocorrerem na tecnologia ou no escopo do probelma. O Struts permite que os web-designers dos sites e os desenvolvedores foquem suas atenções para aquilo que eles sabem realmente fazer de melhor. (DAVIS, 2001)

2.6 O Mapeamento de Objetos para Banco de Dados Relacional

Segundo ZIMBRÃO (2003) para a chamada “transição suave” da análise para o projeto e implementação orientada a objetos é necessário que, além de programar em uma linguagem OO, se armazene as informações em um banco de dados OO. No entanto, a maioria das oranizações dispõe apenas de sistemas gerenciadores de banco de dados relacionais, tais como Oracle, DB2, SQL Server, MySQL e Firebird, entre outros. É necessário então mapear o modelo de classes e objetos para um modelo relacional.

O paradigma de orientação a objetos está baseado em construir aplicações de objetos que possuem dados e comportamentos, já o paradigma relacional é baseado basicamente em armazenamento dados (AMBLER, 2000a).

Segundo ZIMBRÃO (2003), o termo “descasamento de impedância” é usado para ressaltar as diferenças entre os paradigmas: relacional e orientado a objetos. O paradigma OO é baseado em principios de programação e engenharia de software bem aceitos ao longo dos anos, tais como acoplamento, coesão e encapsulamento, enquanto o modelo relacional é baseado em princípios matemáticos da teoria dos conjuntos.

O termo “descasamento de impedância”, para AMBLER (2000a), aparece quando você observa a estratégia de acesso: no paradigma de objetos você navega nos objetos através de seus relacionamentos, já com o paradigma relacional você duplica dados (chaves estrangeiras) para realizar as junções entre as linhas das tabelas. Esta diferença fundamental resulta em uma combinação não ideal dos dois paradigmas, porém quando que duas coisas diferentes não foram utilizadas em conjunto com alguns ajustes? Um dos segredos do sucesso do mapeamento de objetos para banco de dados

relacional é entender os dois paradigmas, e suas diferenças, e fazer então ajustes inteligentes baseados neste conhecimento.

2.6.1 A importância dos Identificadores de Objetos

Segundo AMBLER (2000b), nós precisamos associar um identificador único para os nossos objetos para que possamos identificá-los. Na terminologia relacional um identificador único é chamado de “chave”, na terminologia de objetos é chamado de identificador de objetos (*object identifier* - OID). OIDs são tipicamente implementados como objetos desenvolvidos em suas aplicações orientadas a objetos como inteiros grandes, ou diversos inteiros grandes para aplicações maiores, no seu esquema relacional. Os identificadores de objetos nos permitem simplificar a estratégia de “chave” dentro de uma base de dados relacional. Embora OIDs não resolvam completamente a questão de navegação entre objetos eles a facilitam. Uma outra vantagem é que o uso de OIDs nos deixa em uma posição em que é razoavelmente fácil automatizar a manutenção dos relacionamentos entre objetos. Quando todas suas tabelas possuem “chaves” no mesmo tipo de coluna, neste caso OIDs, torna-se muito fácil de escrever o código genérico tirando vantagem deste fato.

2.6.2 Mapeando Atributos para Colunas

É ressaltado por AMBLER (2000a) que um atributo da classe mapeará zero ou mais colunas em uma base de dados relacional. È importante lembrar que nem todos os atributos são persistentes. Por exemplo, uma classe qualquer pode ter um atributo “*total*” que é usado por suas instâncias para cálculos, mas não é salvo no banco de dados. Além disso, alguns atributos de objetos são também objetos, por exemplo, um objeto “*curso*” tem uma instância de “*livroTexto*” como um atributo, que mapeia em várias colunas no banco de dados (realmente, as possibilidades são que a classe “*livroTexto*” mapeará em uma ou mais tabelas). Uma coisa importante é que essa definição é recursiva: em algum ponto o atributo será mapeado em uma ou mais colunas. É também possível que diversos atributos possam ser mapeados para uma simples coluna da tabela. Por exemplo, uma classe representando um número telefônico

pode ter atributos numéricos para representar cada parte do número telefone (DDI, DDD,....), porém o número telefônico pode ser armazenado como uma coluna simples em uma tabela.

2.6.3 Mapeando Classes para Tabelas

As classes são, freqüentemente, mapeadas para tabelas. Exceto em um banco de dados muito simples, segundo AMBLER (2000b), nunca teremos um mapeamento um-para-um de classes para tabelas.

2.6.3.1 Implementando Herança no Banco Relacional

Essa questão, basicamente, recai em como organizar os atributos herdados dentro de um modelo persistente. Segundo AMBLER (2000b) existem três soluções fundamentais para mapear herança em um banco de dados relacional, e para entendê-las vamos discutir o mapeamento do diagrama de classe apresentado na figura 2-3. Para manter a simplicidade, não serão modelados todos os atributos das classes.

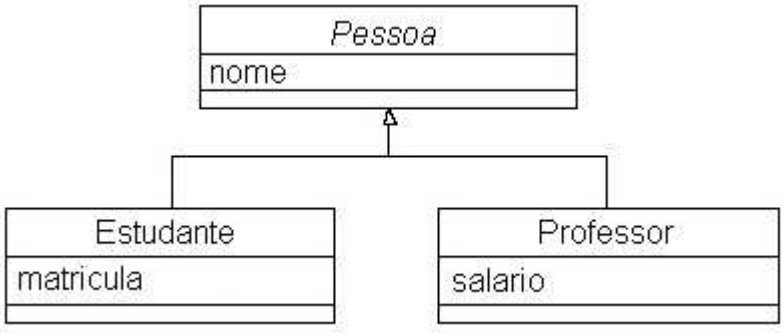

Figura 2-3 Diagrama de classe de uma hierarquia simples

2.6.3.2 Uma Entidade para Todas as Classes de uma Hierarquia

Nessa estratégia, segundo AMBLER (2000a), deve ser mapeada uma hierarquia inteira de classes em uma entidade de dados, onde todos os atributos de todas as classes da hierarquia serão armazenados. A figura 2-4 descreve o modelo de persistência para a hierarquia de classes da figura 2-3. Podemos observar que a coluna "*PessoaOID*" foi

introduzida como chave primária da tabela – nós usaremos OIDs (identificadores sem significado de negócio, também chamadas de chaves artificiais) em todas as soluções.

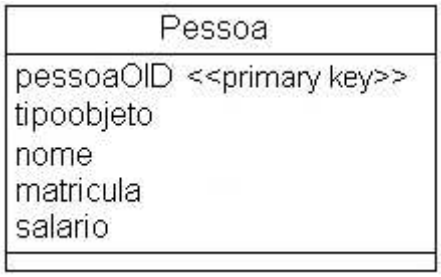

Figura 2-4 Uma tabela para toda hierarquia.

Uma das vantagens, segundo AMBLER (2000a), desta estratégia é a sua simplicidade, onde podemos ressaltar a facilidade na geração de relatórios "ad hoc" (relatório executado para uma proposta específica de um grupo pequeno de usuários, que geralmente escrevem eles mesmos os relatórios), porque todos os dados são encontrados em apenas uma tabela. As desvantagens são que toda vez que um novo atributo é adicionado em qualquer das classes da hierarquia uma nova coluna precisará ser adicionada na tabela. Isto aumenta o acoplamento entre as classes da hierarquia. Caso ocorra algum erro ao adicionar um único atributo, poderíamos afetar todas as classes dentro da hierarquia em qualquer que fosse a classe que ganhasse o novo atributo. Existe também um desperdício potencial de espaço no banco de dados. Temos também que adicionar a coluna "*objectType*" para indicar se a linha representa um estudante, um professor, ou um outro tipo de pessoa. Essa estratégia, normalmente, funciona bem quando temos somente um único papel, mas não quando existem múltiplos papéis (por exemplo, uma pessoa que é estudante e professor ao mesmo tempo).

2.6.3.3 Uma Entidade para Cada Classe Concreta

Nesta estratégia, AMBLER (2000a), ressalta que cada entidade de dados deve conter os seus atributos e os atributos herdados da classe abstrata. A figura 2-5 descreve o novo modelo de persistência para a hierarquia de classe apresentada anteriormente. Existe entidade de dados correspondente tanto para a classe "*Estudante*" quanto para a

classe “*Professor*” porque elas são concretas, mas não para a classe “*Pessoa*” porque ela é abstrata. Cada entidade de dados tem associada sua própria chave primária, “*estudanteOID*” e “*professorOID*” respectivamente.

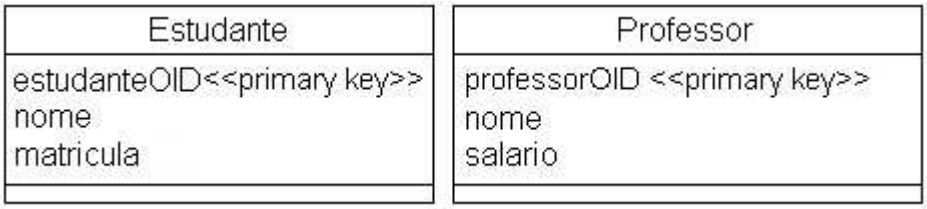

Figura 2-5 Uma entidade de dados para cada classe concreta.

Para AMBLER (2000a), a grande vantagem desta estratégia é que ainda é razoavelmente fácil executar as consultas “*ad hoc*”, dado que todos os dados que você precisa sobre uma única classe estão armazenados em somente uma única tabela. Entretanto, existem muitas desvantagens. Uma é que quando modificamos uma classe temos que modificar sua tabela e as tabelas das possíveis subclasses. Por exemplo, caso precisássemos adicionar “*peso*” e “*altura*” na classe “*Pessoa*”, teríamos que atualizar as duas tabelas. Uma outra desvantagem seria se objeto mudasse seu papel – por exemplo, a instituição que torna professor um dos seus estudantes – neste caso teríamos que copiar os dados para a tabela apropriada e associar um novo OID. Finalizando, é difícil suportar múltiplos papéis e ainda manter a integridade dos dados. Por exemplo, onde armazenaríamos o nome de alguém que é um estudante e um professor?

2.6.3.4 Uma Entidade por Classe

Nesta estratégia, AMBLER (2000a), sugere a criação de uma tabela para cada classe, contendo os atributos identificadores (OID’s) e os atributos que são específicos de cada classe. A figura 2-6 ilustra o modelo de persistência para a hierarquia que estamos utilizando como exemplo. Nela verificamos que o atributo “*pessoaOID*” é utilizado como chave primária para todas as três entidades de dados. Uma característica interessante é que as colunas “*pessoaOID*” tanto em “*Professor*” e “*Estudante*” estão associadas a dois estereótipos, algo que não é permitido na UML.

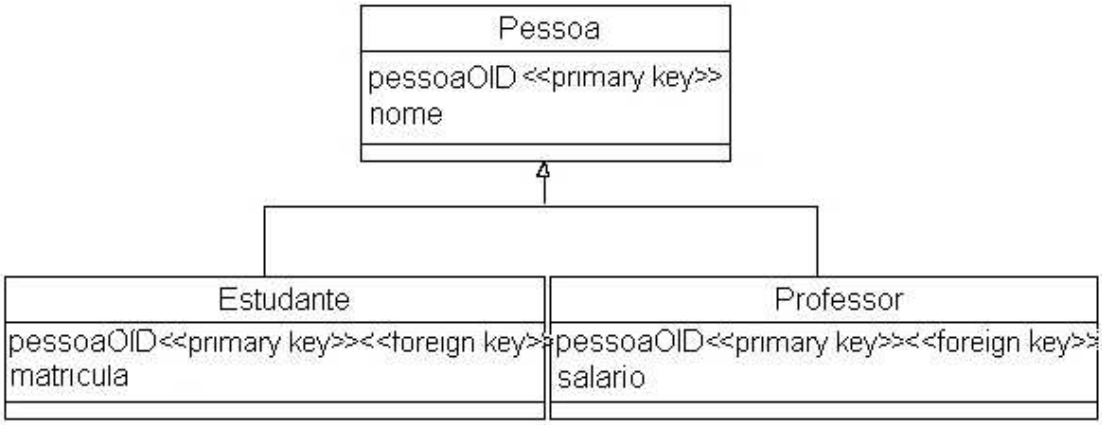

Figura 2-6 Uma entidade de dados por classe.

Para AMBLER (2000a), a vantagem principal desta estratégia é a aderência aos conceitos da orientação a objetos. Ela suporta muito bem o polimorfismo porque temos meramente registros nas tabelas apropriadas para cada papel que um objeto pode ter. É também muito fácil modificar superclasses e adicionar novas subclasses porque precisamos apenas modificar ou adicionar uma tabela. Assim como as anteriores, essa estratégia apresenta algumas desvantagens. A saber: (i) a ocorrência de muitas tabelas no banco de dados – uma para cada classe, de fato (mais as tabelas para manter os relacionamentos); (ii) leva muito tempo para ler e escrever dados usando está técnica, porque temos que acessar múltiplas tabelas. Este problema pode ser suavizado caso as tabelas fiquem fisicamente em discos diferentes (assumindo que as cabeças dos discos operam independentemente); (iii) relatórios "*ad hoc*" no banco de dados são difíceis, ao menos que tenhamos várias visões para simular as consultas desejadas.

Fatores a considerar	Uma tabela por hierarquia	Uma tabela por classe concreta	Uma tabela por classe
Consulta "Ad-hoc"	Simples	Médio	Médio/Difícil
Facilidade de implementação	Simples	Médio	Difícil
Facilidade de acesso aos dados	Simples	Simples	Médio/Simples

Acoplamento	Muito alto	Alto	Baixo
Velocidade de acesso aos dados	Rápido	Rápido	Médio/Rápido
Suporte ao polimorfismo	Médio	Baixo	Alto

Tabela 2-1 Comparando as estratégias de mapeamento de herança.

2.6.4 Mapeando Associações, Agregação e Composição

Não apenas objetos precisam ser mapeados para a base de dados, precisamos, também, mapear os relacionamentos que os objetos estão envolvidos permitindo que eles possam ser restaurados em uma data futura. Segundo AMBLER (2000a) existem quatro tipos de relacionamentos que um objeto pode estar envolvido: herança, associação, agregação, e composição. Para mapear efetivamente esses relacionamentos nós precisamos entender a diferença entre eles, como implementar geralmente relacionamentos, e como implementar especificamente relacionamentos muitos-para-muitos.

2.6.5 A diferença entre associação, agregação e composição

Para AMBLER (2000b), a partir da perspectiva de banco de dados, a única diferença entre relacionamentos de associação e agregação/composição é como os objetos são influenciados pelos outros. Com agregação e composição, qualquer coisa que se faça com o todo no banco de dados você quase sempre precisa fazer às partes, já com associação não é o caso. Na Figura 2-7 você pode ver três classes, duas que tem uma associação simples entre elas e duas que compartilham um relacionamento de agregação (atualmente, a composição provavelmente seria a maneira mais exata de modelar isto). A partir do ponto de vista do banco de dados, agregação/composição e associação são diferentes pelo fato que com agregação você usualmente quer ler uma parte quando você lê o todo, já com associação não é sempre óbvio que precisamos

fazer isto. O mesmo acontece para salvar objetos da base de dados e apagar ojetos da base de dados. Isto é geralmente específico ao domínio do negócio, mas esta regra parece valer para a maioria das circunstâncias.

Figura 2-7 Diferença entre associação e agregação/composição

2.6.6 Implementando relacionamentos em banco relacional

Segundo AMBLER (2000b), relacionamentos no banco de dados relacional são mantidos através do uso de chaves estrangeiras. Uma chave estrangeira é um ou mais atributos de dados que aparecem em uma tabela que pode ser parte de uma chave de outra tabela, ou coincidente totalmente, com a chave da outra tabela. Chaves estrangeiras permitem que você relacione uma linha de uma tabela com uma linha em outra tabela. Para implementar relacionamentos um-para-um e um-para-muitos você simplesmente tem que incluir a chave de uma tabela na outra tabela. Na Figura 2-8 você vê três tabelas, suas chaves (OIDs), e a chave estrangeira usada para implementar os relacionamentos entre elas. Primeiro existe uma associação um-para-um entre as entidades de dados Posição e Empregado. Para implementar este relacionamento usamos o atributo posicaoOID, que é a chave da entidade de dados “Posição”, dentro da entidade de dados “Empregado”. Fomos forçados a fazer deste jeito porque a associação é uni-direcional – linhas de empregado conhecem sobre suas posições, mas não o contrário. Se tivéssemos uma associação bi-direcional, poderíamos também ter adicionado uma chave estrangeira chamada empregadoOID em “Posição”. Segundo, nós implementamos a associação um-para-muitos entre Empregado e Tarefa usando a mesma estratégia, a única diferença é que colocamos a chave estrangeira em Tarefa porque era o lado “N” do relacionamento.

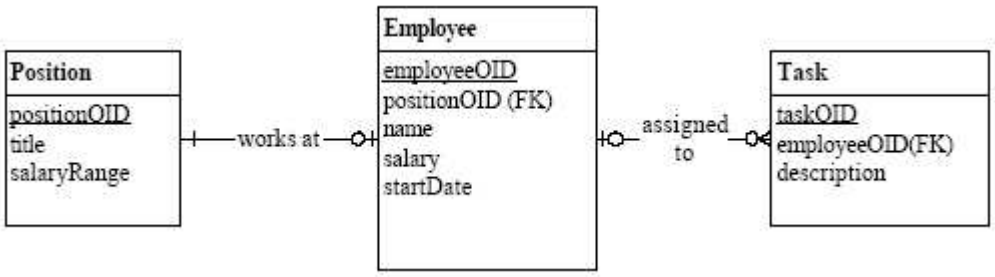

Figura 2-8 Implementando relacionamentos (AMBLER, 2000b).

2.6.7 Implementando associações muitos-para-muitos

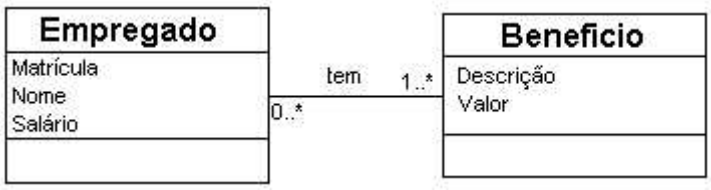

Figura 2-9 Relacionamento muitos-para-muitos.

Para implementar relacionamentos “muitos-para-muitos”, segundo AMBLER (2000a), precisamos do conceito de tabela associativa, que é uma entidade de dados cuja única finalidade é manter a associação entre duas ou mais tabelas em uma base de dados relacional. Na figura 2-9 vemos que existe um relacionamento “muitos-para-muitos” entre Empregado e Beneficio. Na figura 2-10 vemos como usar uma tabela associativa para implementar um relacionamento “muitos-para-muitos”. Em um banco relacional os atributos contidos em uma tabela associativa são tradicionalmente a combinação das chaves das tabelas envolvidas no relacionamento. O nome da tabela associativa é tipicamente ou a combinação dos nomes das tabelas que estão associadas ou o nome da associação que elas implementam. Neste caso foi escolhido “EmpregadoBeneficio” ao invés de “BenefícioEmpregado” pois sentimos que reflete melhor a natureza da associação. Note a aplicação das multiplicidades na Figura 2-10. A regra é que as multiplicidades se cruzem uma vez que é incluída uma tabela associativa, como indicado na figura 2-10. A muliplicidade “1” é sempre introduzida nas bordas exteriores, como você pode ver na figura 2-10, para preservar a multiplicidade total da associação original. A associação original indica que um empregado tem um ou mais

benefícios e que qualquer benefício está associado a zero ou mais empregados. Na figura 2-10 você vê que isto ainda é verdade mesmo com a tabela associativa adicionda para manter a associação.

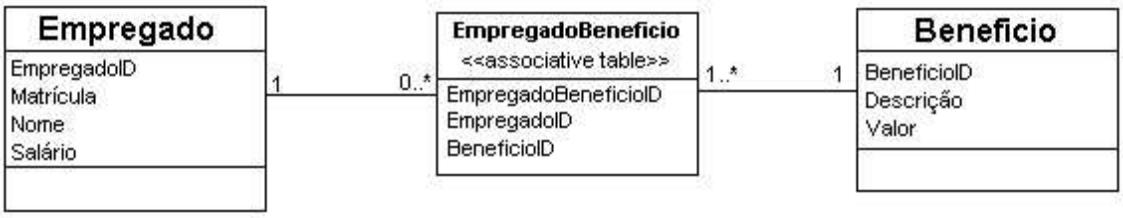

Figura 2-10 Implementação de relacionamento muitos-para-muitos.

É importante notar que AMBLER (2000a) escolheu aplicar o estereótipo "<<tabela associativa>>" ao invés da notação para classes associativas – uma linha tracejada conectando a classe associativa para a associação que ela descreve – por dois motivos. Primeiro, a proposta de uma tabela associativa é de implementar uma associação, já a de uma classe associativa é de descrever uma associação. Segundo, a estratégia tomada na figura 2-10 reflete a estratégia de implementação real que você necessitaria fazer para utilizar a tecnologia relacional.

Para ZIMBRÃO (2003), deve-se ter em mente que, apesar de se estar usando um modelo OO, todos os mecanismos disponíveis no banco de dados relacionais, tais como integridade referencial, unicidade, restrições e validações, devem ser utilizados para garantir o estado válido dos dados. Por outro lado, o que deve guiar um bom mapeamento é o grau de fidelidade ao modelo de classes e objetos, sempre contraposto à razão custo/benefício. Não se deve realizar o mapeamento com um "pensamento relacional", aplicando técnicas de identificação de entidades e normalização de forma cega – o produto final pode ficar tão distante do modelo original que a programação e manutenção se tornarão extremamente difíceis.

2.7 A Metodologia Rápida de Desenvolvimento de Sistemas

Uma metodologia é um agregado de técnicas e ferramentas que tem por objetivo padronizar o processo de desenvolvimento de sistemas em uma empresa (GHEZZI,

JAZAYERI e MANDRIOLI, 1991). Uma metodologia para desenvolvimento de sistemas especifica a seqüência de passos e a serem seguidos durante o desenvolvimento de um sistema de informação. A cada um destes passos, associa-se um conjunto de atividades, seus produtos e as regras de verificação que garantem a passagem para a próxima fase (PRESSMAN, 2001).

Os produtos das diferentes fases de uma metodologia são especificações que descrevem, com um certo grau de abstração, o sistema de informação que está sendo desenvolvido. Esta especificação abstrata é chamada de modelo. Utilizamos modelos na construção de sistemas complexos, porque em geral não conseguimos compreender o sistema em sua totalidade. Um modelo é uma descrição da realidade que ressalta alguns aspectos em detrimento de outros. Cada tipo de modelo utiliza uma notação precisa e um conjunto de regras sintáticas e semânticas (SCHMITZ e SILVEIRA, 2000).

Um dos objetivos da Metodologia Rápida de Desenvolvimento de Sistemas (MRDS), segundo SILVEIRA e SCHMITZ (2001) é diminuir a quantidade de documentos produzidos na construção de um sistema de qualidade (ver figura 2-11). Desse modo, dividimos o processo de desenvolvimento em duas fases. A primeira é a definição do Modelo de Requisitos. Essa fase tem por objetivo definir os requisitos do sistema, ou seja, responder a seguinte pergunta: qual as funcionalidades esperadas pelo sistema? Não é escopo desse artigo definir como obter um bom modelo de requisitos. Entretanto, o modelo de requisitos é fundamental para se chegar a um bom projeto.

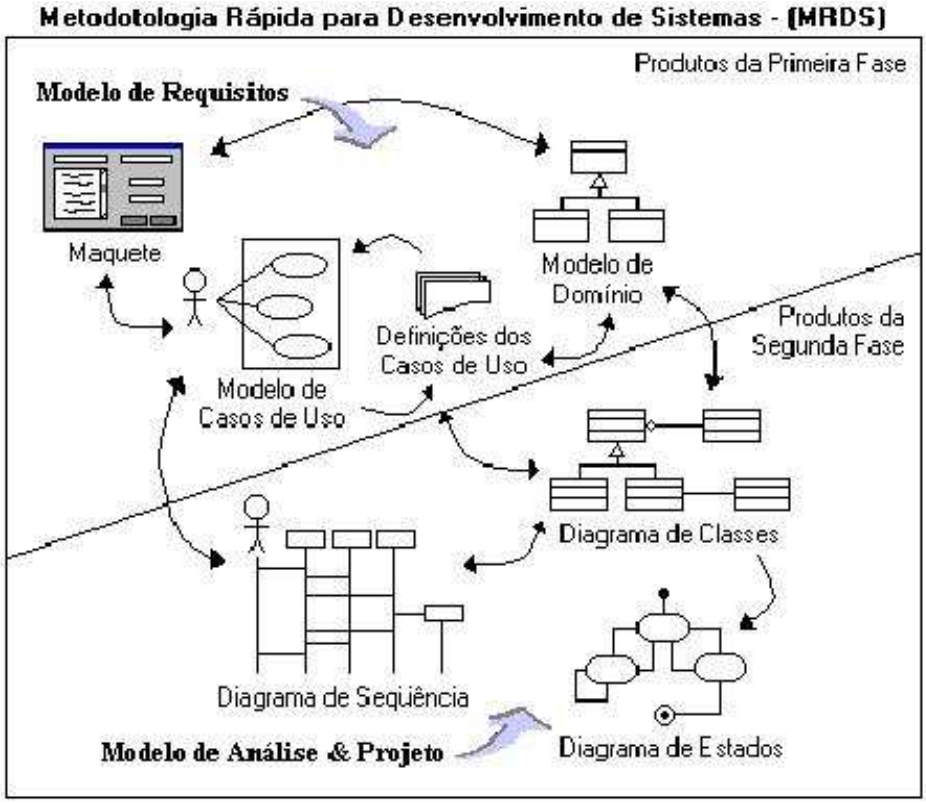

Figura 2-11 Diagrama esquemático da MRDS.

A segunda fase, chamada de Modelo de Análise e Projeto, tem como objetivo transformar o Modelo de Requisitos na definição dos módulos componentes do sistema. Nessa fase fazemos uso de uma arquitetura padrão. A arquitetura padrão utilizada não especificará uma solução particular em detrimentos de outra, mas guiará o projetista a uma forma de estruturar suas decisões que o levará a um resultado positivo. (SILVEIRA e SCHMITZ, 2001)

2.7.1 Modelo de Requisistos

A figura 2-12 mostra um meta-modelo para exemplificar a estrutura do Modelo de Requisitos. Nela podemos verificar que o Modelo de Requisitos é composto pelos: Modelo de Casos de Uso, Modelo de Domínio e as Maquetes[3] do sistema. Por sua vez, o Modelo de Casos de Uso é composto pelos Casos de Uso, que devem ser estruturados em pacotes afins. Por fim, cada Caso de Uso deve ter associado a ele a sua definição.

(SILVEIRA e SCHMITZ, 2001)

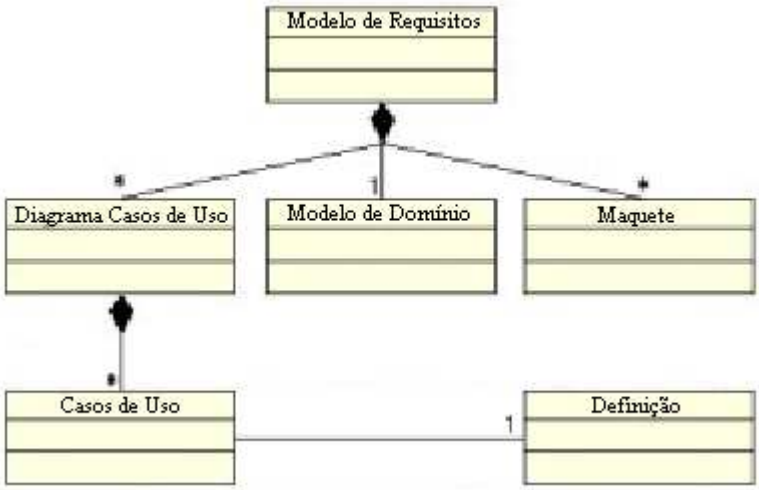

Figura 2-12 Meta-modelo do Modelo de Requisitos.

Em SILVEIRA e SCHMITZ (2001) é apresentado o roteiro a ser seguido na construção do Modelo de Requisitos, e segue os seguintes passos:

1. Definir o escopo do sistema: isto pode ser feito inicialmente através de um documento inicial de requisitos, na forma de texto;

2. Criar uma versão preliminar do Modelo de Domínio, ou seja, as classes do domínio do problema, definindo quais as entidades e suas definições. Este modelo será melhorado e corrigido a medida que os casos de uso forem sendo detalhados.

3. Executar iterativamente as seguintes etapas:

    a. Listar os casos de uso e os atores que interarem com o sistema;

    b. Acrescentar a cada um dos Casos de Uso;

        i. uma maquete da interface; e

        ii. uma definição para o caso de uso. Essa definição deve seguir

o modelo utilizado em SCHMITZ e SILVEIRA (2000);

c. Estruturar os casos de uso;

   i. Definir as extensões para os casos; e

   ii. Definir os casos que podem ser colocados em evidência e reaproveitados;

d. Definir os pacotes de casos de uso;

4. Verificar o modelo para tirar os possíveis erros de consistência entre as maquetes e a descrição dos casos de uso.

2.7.2 Modelo de Análise e Projeto

O objetivo desta fase, para SILVEIRA e SCHMITZ (2001), é produzir um plano que permita a construção do sistema. Essa fase é composta pelos Modelos Estático e Dinâmico. O Modelo Estático é composto por um conjunto de diagramas de classes. O Modelo Dinâmico é composto por um conjunto de diagramas de seqüência e de estados. A figura 2-13 apresenta o meta-modelo relativo ao Modelo de Análise e Projeto.

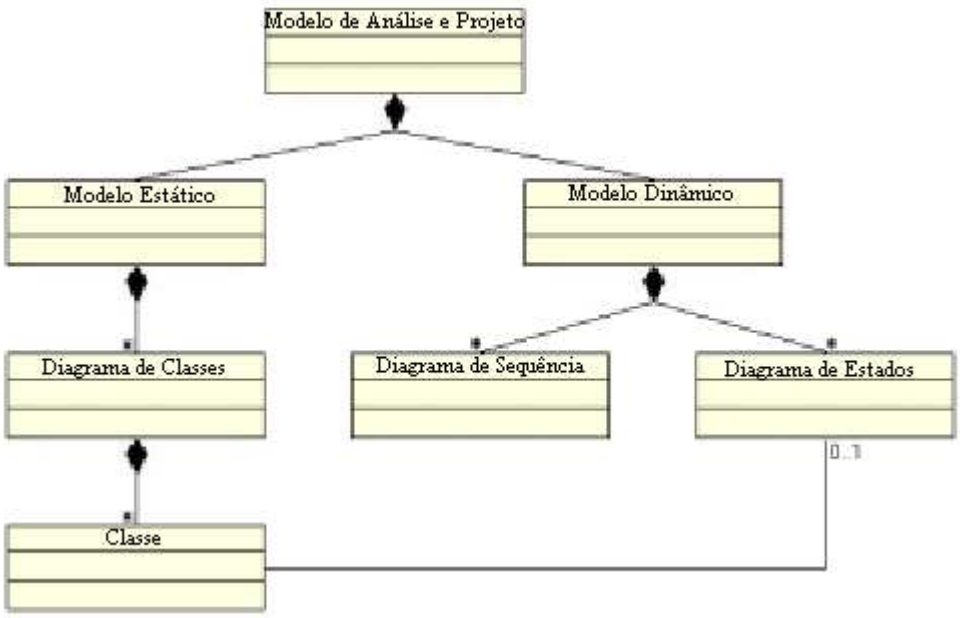

Figura 2-13 Meta-modelo do Modelo de Análise e Projeto.

Segundo SILVEIRA e SCHMITZ (2001), esta etapa deve ser feita de forma iterativa pelo projetista, o que significa que ele não deve esperar obter o modelo final na primeira vez, mas sim, que terá de executar estes passos repetidamente até obter um modelo completo e consistente. A figura 2-11 mostra o esquema da interação das fases da MRDS. O roteiro para a construção deste modelo é:

1. Para cada pacote de casos de uso definido no Modelo de Requisitos, deverá ser definido um diagrama de classes. Esse diagrama de classes deverá mostrar os relacionamentos entre as classes de domínio, interface e controle. Na próxima seção apresentaremos com mais detalhes essas classes.

2. Para cada pacote de casos de uso definido no Modelo te Requisitos, deverá ser definido um diagrama de seqüência. Este passo pode ser tornado de rotina se lembrarmos que as interações entre o objeto de controle e os objetos de interface ou de domínio seguem alguns padrões definidos em SCHMITZ e SILVEIRA (2000); A criação dos diagramas de seqüência é um passo fundamental para a definição da implementação dos casos de uso do sistema. Ao definir os serviços que cada um dos objetos participantes do caso deve prover para a execução do mesmo, o projetista vai refinando as definições das classes do sistema, previamente, definidos no Modelo de Domínio.

3. Criar um diagrama de transição de estados para as classes de domínio, interface e controle que tenham ciclos de vida não-triviais;

4. Examinar novamente os diagramas de classes para verificar se algumas relações de herança podem ser descobertas; e

5. Verificar os modelos para tira os eventuais erros de consistência entre os modelos definidos.

A MRDS vem sendo desenvolvida desde 1997. Neste período ela vem sendo aprimorada pelo uso, tanto em turmas de graduação como em projetos de empresas de pequeno e médio porte. (SILVEIRA e SCHMITZ, 2001)

A adoção de uma metodologia de desenvolvimento de sistemas em empresas de pequeno e médio porte encontra como obstáculos: (1) o custo da implantação, (2) o impacto do uso da metodologia na produtividade dos profissionais da empresa. Em SILVEIRA e SCHMITZ (2001) é apresentado que a MRDS permite atacar estes dois problemas da seguinte maneira:

- Custo: a MRDS reduz o tempo gasto no ensino da metodologia devido a sua simplicidade; e

- Impacto: o foco da MRDS é na utilização dos modelos que são diretamente empregados na geração de código-fonte. Desta maneira, o tempo de modelagem é efetivamente tempo de projeto e (indiretamente) de codificação.

2.8 Os Diagramas da UML Utilizados

A construção de sistemas é centrada em modelos. Os modelos são documentos expressos em uma linguagem determinada, geralmente gráfica, que contêm semânticas interpretáveis de forma fácil para quem os lê. Através dos modelos, conseguimos obter múltiplas visões do sistema, particionando a complexidade para sua compreensão e atuando como meio de comunicação entre os participantes do projeto. Deste modo, uma linguagem de modelagem padronizada é fundamental para a construção e o entendimento de bons modelos.

Uma linguagem deve conter elementos de modelagem que representem os conceitos e semânticas fundamentais, uma notação para visualização gráfica dos elementos de modelos e regras e conselhos de como usá-la. A sintaxe define as construções existentes e como elas compõem outras, de forma independente de notação. As semânticas definem como as instâncias das construções devem se relacionar com

outras instâncias, de forma a exibir algum significado bem formado (OMG, 2004).

A partir de 1994, Grady Booch, Jim Rumbaugh e Ivar Jacobson iniciaram a unificação dos seus métodos, que já despontavam dentre os métodos orientados a objetos existentes na época. Desta unificação, foi criada a *Unified Modeling Language* (UML), posteriormente aceita pela OMG como a linguagem padrão (1997) e estando, neste momento, na versão 1.5. (OMG, 2004):

"A Unified Modeling Language (UML) é uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software, assim como para modelagem de negócios e outros tipos de sistemas. A UML representa uma coletânea das melhores práticas de engenharia que se mostraram vitoriosas na modelagem de sistemas grandes e complexos". (OMG, 2004)

A seguir apresentaremos uma breve descrição dos principais elementos e diagramas que formam o subconjunto utilizado no escopo deste trabalho. A MRDS utiliza apenas um subconjunto da UML.

2.8.1 Diagrama de Casos de Uso

Os diagramas de casos de uso foram sugeridos pelo método OOSE de JACOBSON *et al.* (1992) são uma forma de representar os requisitos funcionais de um sistema em um alto nível de abstração, sem considerar os objetos que fazem parte do sistema, a estrutura de classes e detalhes implementacionais. Neste sentido, indicam um caminho para a análise e compreensão do sistema como um todo sem a preocupação com o seu funcionamento interno. Ao utilizar notação simples e linguagem natural, esses diagramas criam uma forma clara e simples de comunicação e especificação entre usuários e membros da equipe de desenvolvimento. Os elementos participantes são os casos de uso, atores e seus relacionamentos (BOOCH, JACOBSON e RUMBAUGH, 1999).

Para BOOCH, JACOBSON e RUMBAUGH (1999), um caso de uso é a especificação de um determinado serviço esperado pelo sistema, composta por uma

seqüência de ações e variações que representam os cenários de interação a partir dos eventos externos iniciais. Cada cenário representa uma seqüência de realização de um caso de uso determinado. Cada caso de uso pode ser posteriormente refinado em uma ou mais classes de objetos e um conjunto de colaborações entre eles para a realização da funcionalidade esperada. É possível a utilização de pontos de extensão e inclusão de referências a outros casos de uso dentro de um determinado caso de uso. Esses relacionamentos são expressos por ligações especiais com estereótipo de uso e extensão, além dos relacionamentos de herança que também podem fazer parte deste diagrama.

Um ator, segundo BOOCH, JACOBSON e RUMBAUGH (1999), é a representação de uma entidade do mundo externo que atua diretamente sobre o sistema, interagindo com ele de acordo com um papel definido e não apenas um indivíduo. Um ator pode se comunicar com casos de uso, através de uma ligação de comunicação, que reflete sua participação neste caso. É também possível o uso de heranças entre atores.

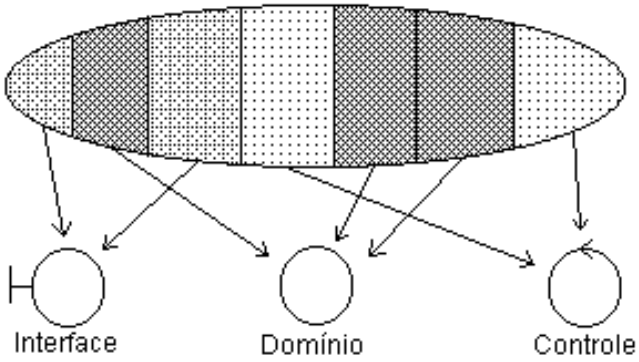

Figura 2-14 Particionamento das funcionalidades do Caso de Uso

Cada caso de uso é inteiramente dividido nos três tipos de classes descritas acima. Uma classe pode, obviamente, ser comum a vários casos de uso. Na verdade, é desejável que as classes sejam reutilizáveis em vários casos de uso diferentes, isto sugere que eles também serão reutilizáveis em mudanças futuras do sistema (JACOBSON *et al.* 1992).

2.8.2 Diagrama de Classes

Como visto em BOOCH, JACOBSON e RUMBAUGH (1999), o diagrama de classe representa a estrutura estática mais importante da modelagem UML, sendo formados por classes de objetos com características e comportamento semelhantes e seus relacionamentos mais comuns, como associações, agregações, heranças e dependências.

O principal elemento de modelagem deste diagrama estrutural é a classe, que representa a descrição de um conjunto de objetos envolvidos no sistema. Objetos apresentam um estado, exibem algum comportamento bem definido e possuem uma identidade única que os diferenciam dos demais objetos. O estado de um objeto é caracterizado pelo seu conjunto de propriedades estáticas e seus valores dinâmicos. O seu comportamento é caracterizado pela atividade do objeto em função das mudanças de estados e troca de mensagens (BOOCH, JACOBSON e RUMBAUGH, 1999).

A utilização de um método OO, por si só, não resulta em uma estrutura mais robusta para futuras mudanças. A construção do *Modelo de Análise e Projeto* tem como objetivo capturar três dimensões do sistema: a informação, o comportamento e apresentação. Muitos métodos de análise orientada a objetos escolhem ter um tipo de objeto apenas que representa qualquer das três dimensões. Nós vamos utilizar três tipos de objetos. O resultado disso é proporcionar uma estrutura que será muito mais adaptável a mudanças. Os tipos de classes utilizadas serão as classes de domínio, classes de interface e classes de controle (JACOBSON *et al* 1992)

Figura 2-15 Instâncias das respectivas classes (PINTO e BASTOS, 2000)

2.8.3 Diagrama de Seqüência

Os diagramas de seqüência, segundo BOOCH, JACOBSON e RUMBAUGH (1999), são um tipo de diagrama de interação, que exibem o comportamento dinâmico do sistema, focalizando a interação entre objetos através da representação da troca de mensagens entre eles para a realização de determinada tarefa ou funcionalidade especificada em um caso de uso. Os diagramas de colaboração diferem do de seqüência apenas em função da sua organização e do enfoque desejado.

Enquanto os primeiros favorecem a visão simples e plana dos objetos e suas interações, os outros favorecem uma visão seqüencial de como as coisas acontecem ao longo do tempo. Os principais participantes do diagrama de seqüência são objetos e mensagens (BOOCH, JACOBSON e RUMBAUGH, 1999).

Um objeto, conforme visto anteriormente, é uma instância de uma classe, que possui características e comportamento (serviços) bem definidos e representa um papel na colaboração sendo modelada. Cada objeto, para BOOCH, JACOBSON e RUMBAUGH (1999), possui uma linha de vida que indica o ciclo de vida do objeto durante uma interação.

Mensagens, como visto em BOOCH, JACOBSON e RUMBAUGH (1999), representam uma comunicação entre duas instâncias em uma interação. Cada mensagem está associada a um papel remetente e um destinatário (instâncias de classificadores definem papéis em uma colaboração), além de possuir uma ação que envia um estímulo (um sinal ou uma chamada de operação). Há estímulos especiais que definem a criação ou destruição de objetos em um diagrama de seqüência. Os estímulos também podem ser assíncronos ou síncronos e a ordem vertical das mensagens indica como elas se sucedem ao longo do tempo.

Hartson e Hix mostraram em seu estudo (SCHMITZ e SILVEIRA, 2000), diferentes estratégias para alocação do controle do fluxo de sistema em quatro categorias a seguir: (1)***Dominado por computação ou embutido***: o controle da funcionalidade fica com os objetos de domínio; (2)***Dominado pela interface*** **:** o

controle da funcionalidade fica com os objetos de interface. Este é o caso que vem sendo utilizado pelos programadores utilizando linguagens visuais; (3)M***isto*****:** o controle é dividido entre os objetos de controle e os de interface. Requer grande disciplina para não degenerar; (4) ***Balanceado*****:** o controle é mantido separado dos objetos de interface e de computação - tipicamente nos objetos de controle . É o caso do nosso modelo canônico apresentado acima.

O casos 1 e 3 levam a uma estrutura de ***controle descentralizada***, enquanto que as estratégias balanceada e dominado pela interface levam a uma forma de ***controle centralizada***.

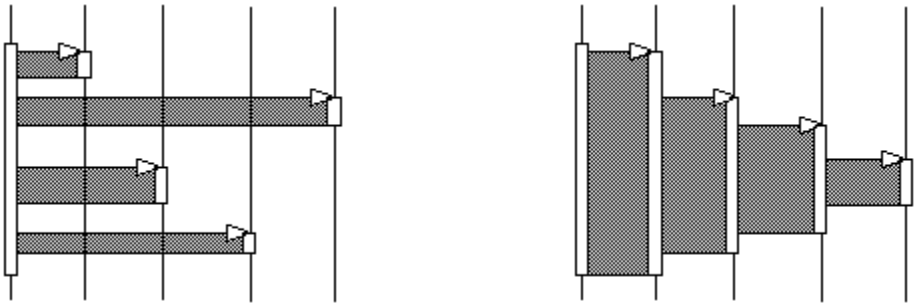

Figura 2-16 Estrutura centralizada e estrutura descentralizada

Os critérios para decisão da melhor forma a ser adotada, segundo SCHMITZ e SILVEIRA (2000), para o diagrama de Seqüência do *Caso de Uso* deve levar em conta as seguintes características do problema.

Se houver uma forte estruturação entre os objetos entidades, então a estrutura conselhada deve ser a descentralizada, já que o próprio problema define a ordem de execução entre os objetos. Se o roteiro do *Caso de Uso* não estiver bem definido, ou for passível de alteração, então a estrutura balanceada (centralizada) deve ser preferida (SCHMITZ e SILVEIRA, 2000).

2.8.4 Diagrama de Transição de Estados

Para BOOCH, JACOBSON e RUMBAUGH (1999), os diagramas de estados são utilizados para representação do comportamento discreto de um sistema através de máquinas de estado finitas, que podem ser ilustradas como dispositivos que recebem um número finito de estímulos como entrada e conseguem processá-las e oferecer respostas. Desta forma, é possível modelar o comportamento de vários elementos através dos seus estados e os estímulos (transições) que participam da máquina de estados. Este diagrama é composto principalmente por estados, pseudo-estados e transições.

Um evento é uma ocorrência observável que ocorre em um instante de tempo. Os tipos de evento são Sinal (signal), Chamada (call), Tempo (time) e Mudança (change). Os eventos de chamada estão relacionados com operações (BOOCH,

JACOBSON e RUMBAUGH, 1999).

Uma ação é a representação da abstração de um procedimento computacional através de envio de mensagens, que causa mudanças no estado do modelo. As ações podem ser de Construção (create), Destruição (destroy), Chamada (call), Retorno (return), Envio (send), Término (terminate) ou Não-Interpretada (uninterpreted). A cada ação de criação corresponde a instanciação de uma classe, enquanto a cada chamada corresponde uma operação e a cada envio, um sinal. Um sinal representa um estímulo assíncrono, uma comunicação entre duas instâncias de elementos (BOOCH, JACOBSON e RUMBAUGH, 1999).

Um estado, segundo BOOCH, JACOBSON e RUMBAUGH (1999), é uma condição ou situação temporária de um objeto que satisfaz alguma condição, espera algum evento ou responde a algum estímulo como resultado de uma evolução de seqüência de estímulos passados. Um estado pode ser simples, final ou composto, neste caso, possui subvértices (estados ou pseudo-estados). Um estado final representa o fim de execução de um estado composto ou de uma máquina de estados. Todo estado possui um compartimento de eventos que são executados quando se entra ou sai do estado ou durante sua permanência. Há também uma lista de transições internas, ou seja, transições cujo efeito não causa a saída do estado atual. Pseudo-estados representam vértices especiais como estados iniciais, vértices de junção (join) e disjunção (fork), entre outros.

Uma transição é um relacionamento entre dois vértices (estados ou pseudo-estados) que indica a mudança de um estado origem para outro destino em resposta a uma instância de um evento. A cada transição há um efeito ou ação associada, controlada ou não por condições de guarda (BOOCH, JACOBSON e RUMBAUGH, 1999).

2.9 O FAST CASE

O FAST CASE foi projetado para proporcionar um uso simples e eficiente. O FAST CASE trabalha com modelos de sistemas utilizando a UML como linguagem de

modelagem. O conjunto de modelos correspondentes a um sistema é chamado de projeto. Cada projeto é armazenado em um diretório distinto, ou seja, não é possível armazenar informações de dois projetos em um mesmo diretório. (SILVEIRA, 1999)

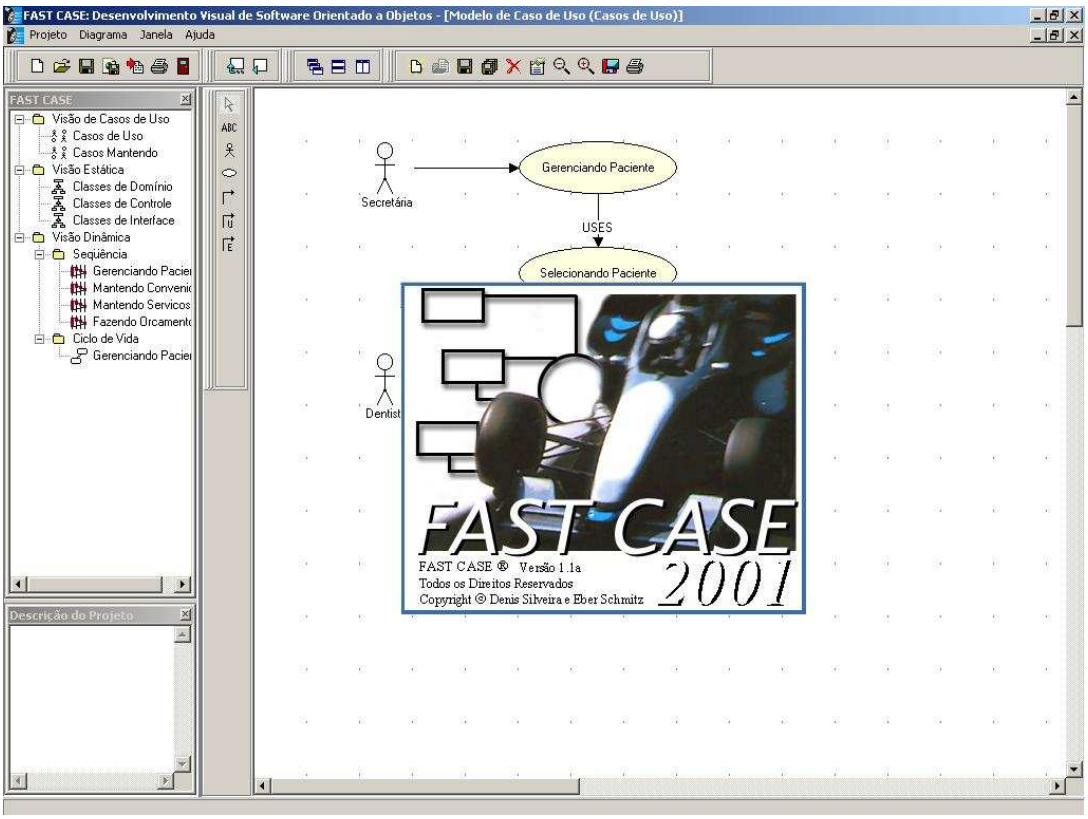

Figura 2-17 O Fast Case.

O FAST CASE suporta a construção e manutenção dos seguintes diagramas da UML: Casos de uso, Classes, Seqüência e Estado. (SILVEIRA, 1999) Na figura 2-17 temos a tela principal da ferramenta FAST CASE.

Com o FAST CASE os alunos podem ter o apoio de uma ferramenta que suporta os conceitos da orientação a objetos nas fases de análise e projeto de um sistema. (SILVEIRA, 1999)

3 O Gerador de Programas

O GP possui uma interface própria com o usuário, permitindo que haja uma interação durante o processo de utilização, independente da ferramenta FAST CASE. Na figura 3-1 é apresentada essa interface.

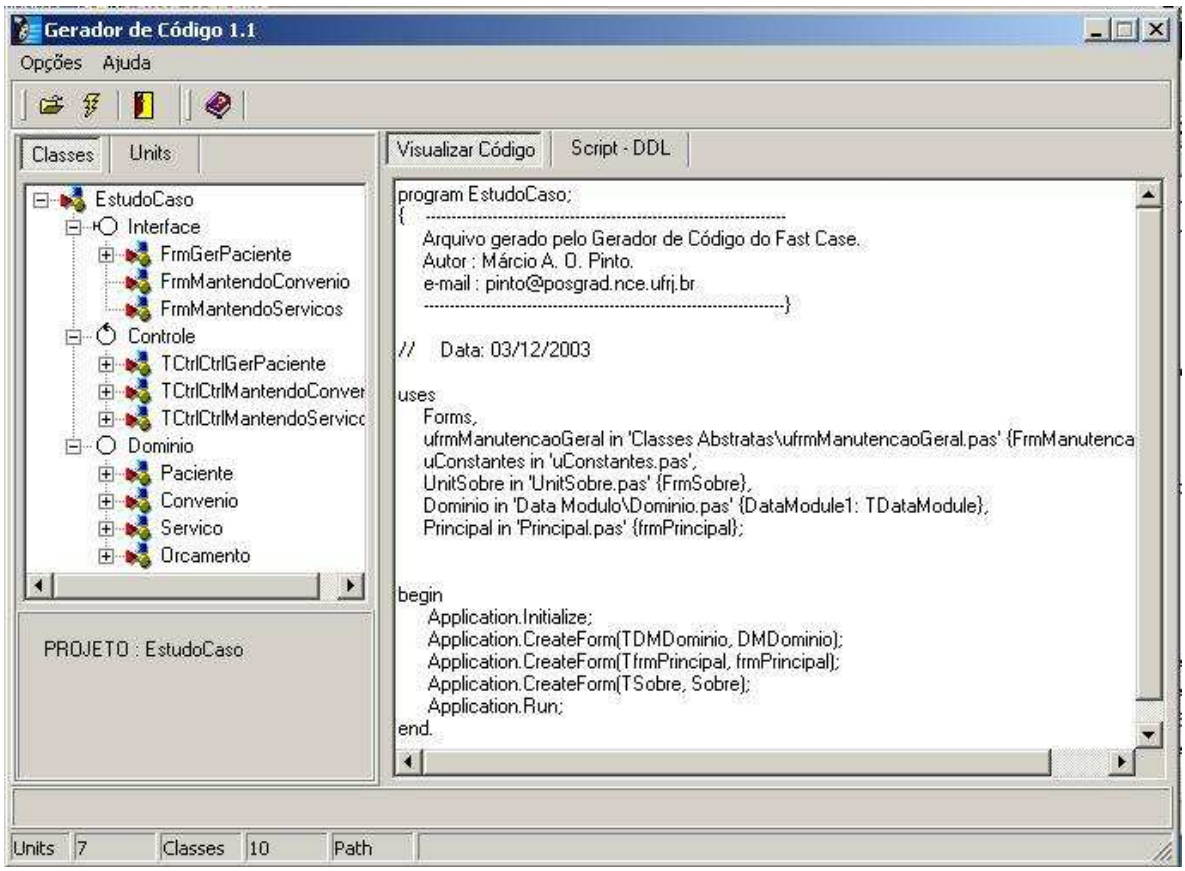

Figura 3-1 Interface principal do Gerador de Programas.

3.1 A arquitetura esperada.

Um dos grandes desafios das metodologias de desenvolvimento de software é resolver o problema de transformação do modelo de requisitos na especificação dos módulos constituintes do sistema.

Esta atividade, conhecida como projeto, é intrinsecamente complexa, pois para

cada modelo de requisitos temos uma infinidade de maneiras de definir os módulos que irão compor a implementação. Uma das maneiras de atacar este problema é definindo uma arquitetura de software padrão para o sistema (SHAW e GARLAN, 1996) (GARLAN *et al*, 1997).

A arquitetura padrão da MRDS, descrita em SCHMITZ e SILVEIRA (2000), define uma infraestrutura para a construção de sistemas orientados a objetos. O objetivo principal dessa arquitetura padrão é prover um ambiente confiável e de fácil manutenção, onde aplicações possam cooperar e evoluir de acordo com a necessidade da organização. A adoção dessa arquitetura padrão faz com que o projetista possa concentrar o seu esforço na solução dos problemas de projeto relevantes ao sistema em desenvolvimento que são: (1) quais as classes que comporão o sistema? (2) quais os serviços que cada uma delas deve prover?

O estilo de arquitetura de software orientado a objetos fornece outras abstrações para projetos de software. Na sua forma mais simples, os objetos permitem encapsular dados e comportamentos em componentes que apresentam uma interface explícita para outros componentes. Neste estilo é possível definir uma interação entre um grupo de objetos, que virão a formar um componente mais complexo. Sendo assim, a arquitetura utilizada pela MRDS, segundo SCHMITZ e SILVEIRA (2000), define três aspectos importantes para os objetos de um sistema:

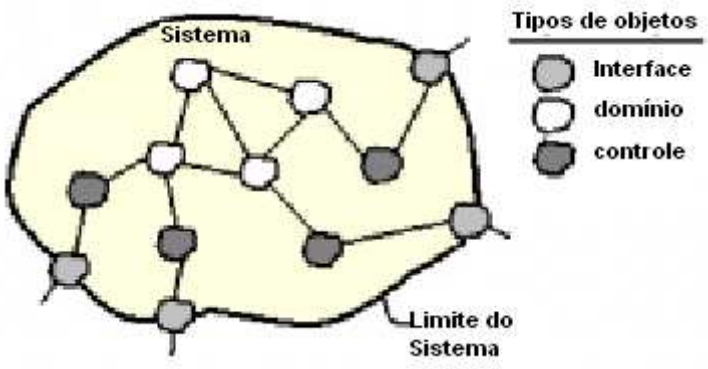

Figura 3-2 As classes da arquitetura padrão (SILVEIRA e SCHMITZ, 2001)

1. os mecanismos utilizados para o armazenamento de dados;

São os objetos que persistem as informações do sistema. Estes objetos correspondem aos objetos do domínio do problema, que são identificados no modelo de Requisitos.

2. os mecanismos utilizados para a interface com o usuário;

São os objetos encarregados da interação entre usuários e sistema. Transformam eventos externos em sinais internos e eventos internos em sinais para o mundo externo;

3. os mecanismos utilizados para o controle do sistema.

São os objetos encarregados do sequenciamento entre os objetos de interface com os objetos de domínio.

3.2 O Mapeamento dos modelos UML

A figura 3-3 apresenta uma visão global, de uma forma macro, do mapeamento dos modelos nos artefatos gerados que o Gerador de Programas realiza. Acrescentando um valor semântico aos modelos da UML.

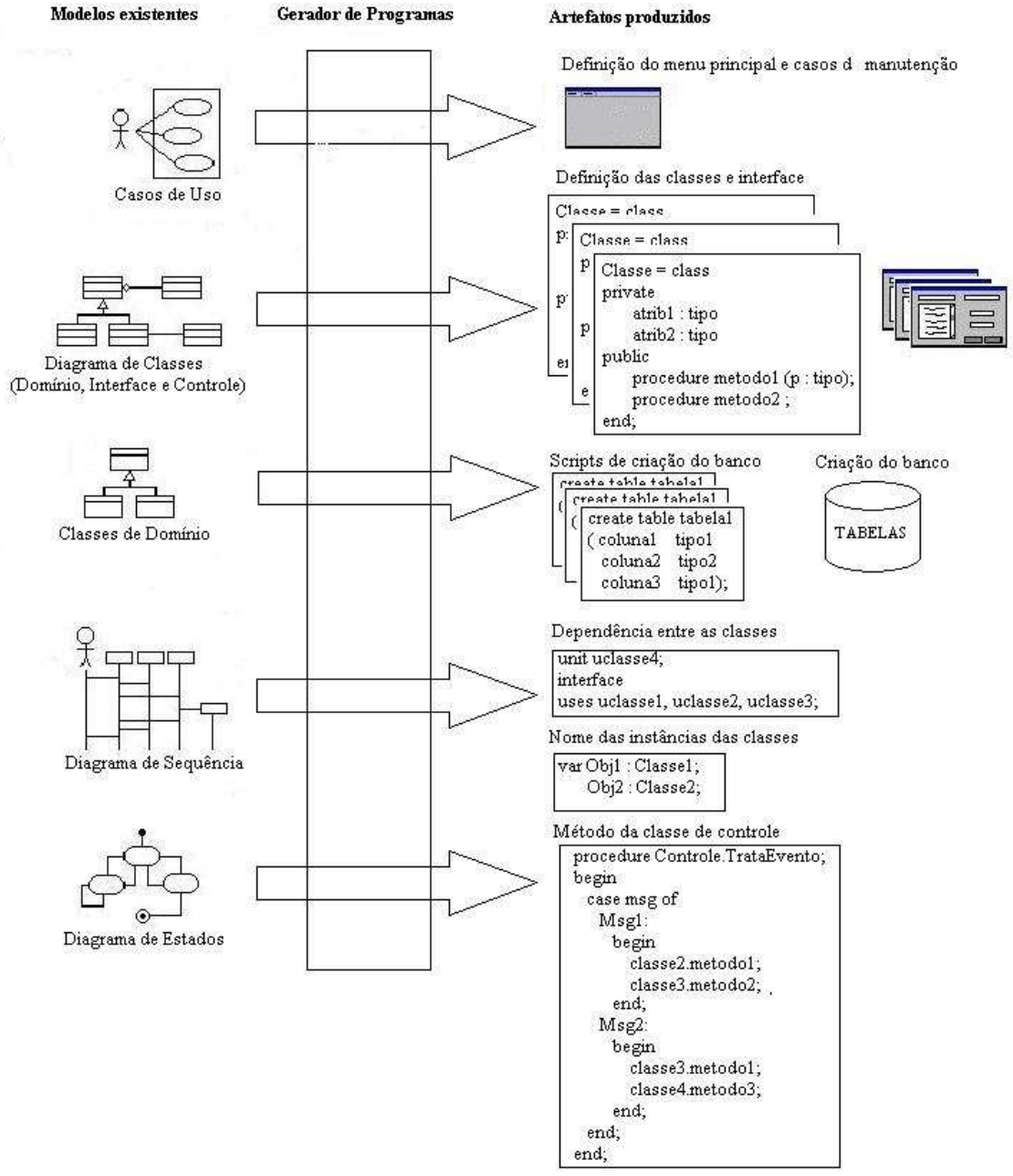

Figura 3-3 Visão global do mapeamento feito pelo GP.

3.2.1 Os Casos de Uso do tipo manutenção

Existem alguns Casos de Uso padronizados, onde os serviços básicos são pré-conhecidos e possuem um comportamento padrão. Esses casos, que foram denomindados de casos de uso do tipo “mantendo”, são os casos de uso que possuem basicamente os serviços de inclusão, exclusão e alteração de dados.

Existe um tratamento especial dado a esses casos do tipo “mantendo”, onde classes de interface e classes de controle referentes a eles serão gerados e tratados de forma particular e única, sempre de uma forma padronizada e são apresentados nas seções sequintes.

3.2.2 Armazenamento de dados: Classes do domínio

Cada classe do domínio do problema representa um conjunto de elementos. No esquema de mapeamento devemos considerar pelo menos duas classes, de forma a representar tanto o conjunto como os seus elementos individuais. Desta forma, um objeto do domínio do problema fica implementado por três classes: (SCHMITZ e SILVEIRA, 2000)

- ***TGerenteObjeto***: uma instância desta classe atua como gerente do conjunto de objetos, atendendo a serviços relativos ao conjunto de elementos. Essa classe é responsável pelos serviços que envolvem procuras, buscas, consultas em todas as instâncias. Entre estes serviços podemos destacar: serviços que respondem a consultas que envolvem todos ou parte dos elementos; serviços que requisitam a criação de um objeto que representa um elemento do conjunto, por exemplo: instanciar o aluno cujo número de registro é X e serviços de manutenção da tabela do banco de dados onde os elementos do conjunto são armazenados.

- ***TTabObjeto*** : esta classe representa a estrutura de dados que faz o acesso aos dados em memória permanente. Note, na figura 3-4 a relação de composição da classe *TGerenteObjeto* com a classe *TTabObjeto*. No ambiente ***Delphi***, por exemplo, a classe *TTabObjeto* pode ser implementada por um componente do tipo *TTable*.

- ***TUmObjeto***: cada objeto desta classe representa um elemento do conjunto. Este objeto é importante quando desejamos manipular um único elemento do conjunto, por exemplo, para sua criação ou alteração. Os principais serviços deste tipo de objetos são: ***1***- oferecer a capacidade de ler e alterar os valores dos atributos mantendo o encapsulamento. ***2***- oferecer a possibilidade de acessar a outros objetos associados via os atributos referenciais que agora se tornam ponteiros para os objetos associados.

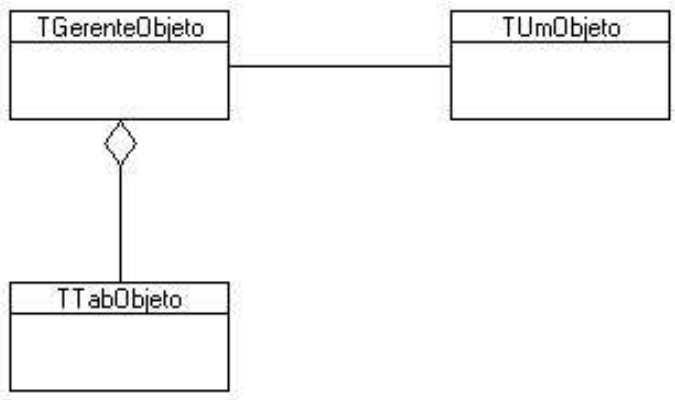

Figura 3-4 Forma Canônica para Mapeamento de Classes de Domínio (SCHMITZ e SILVEIRA, 2000)

O código da unidade de domínio a ser gerado possui apenas uma classe do tipo TDataModule[4], que é constituído por uma classe do tipo TdataBase[2]. Cada classe de domínio gerente é mapeada herdando de uma classe chamada TDominioPai[2], que por sua vez possui uma classe do tipo TTable[2] em sua região privada, e uma classe do tipo TDataSource[2] na área pública, além do construtor da classe, e do método Free.

[4] Ver apêndice B.

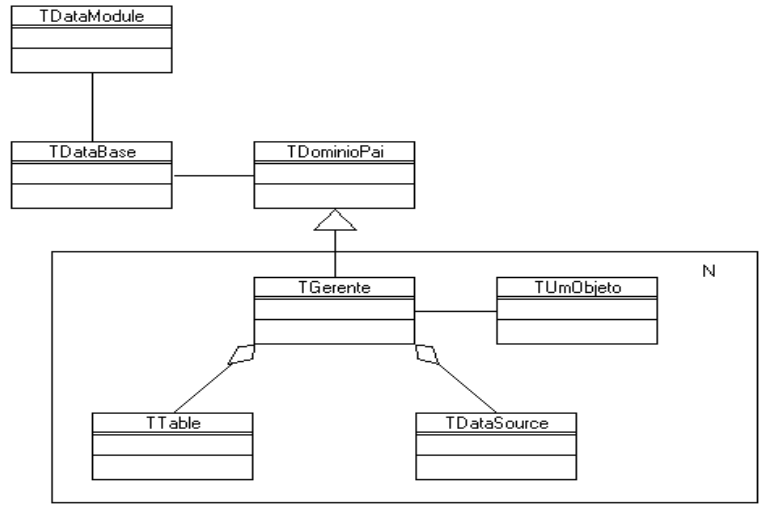

Figura 3-5 Estrutura básica do domínio(PINTO e BASTOS, 2000)

Para cada classe de domínio existente no modelo, é gerada uma tabela no formato Paradox, que será o destino das informações que precisam de persistência. Essa tabela é criada com todos os atributos das classes de domínio correspondente, respeitando o tipo de cada atributo. Na unit de Domínio, é declarada uma constante para cada tabela física existente:

```
const
  tabelaTSocio = 'TSocio.db';
  tabelaTFita = 'TFita.db';
  tabelaTFilme = 'TFilme.db';
  tabelaTReserva = 'TReserva.db';
  tabelaTEmprestimo = 'TEmprestimo.db';
```

Figura 3-6 Constantes com os nomes dos arquivos paradox

A classe TDominioPai, da qual todas as classes gerentes de domínio irão herdar, tem o corpo dos seus métodos gerados de maneira que tudo aquilo que era padrão por todas as classes gerentes ficasse nessa classe (PINTO e BASTOS, 2000).

```
TDMDominio = class(TdataModule)
  DataBase: TdataBase;
  Procedure dmDominioCreate(Sender: Tobject);
```

```
TDominioPai = class
  Private
    { Private declarations }
```

Private { Private declarations } public { Public declarations } end;	tabela : TTable; public { Public declarations } DSource : TDataSource; constructor Create; procedure Free; end;

Figura 3-7 Classes Básicas do domínio.

Para cada classe gerente, é criada uma classe "TUmObjeto", essa classe trata de métodos que são particulares a um único objeto, e a classe gerente fica com os métodos que são para um conjunto de objetos daquele tipo.

TTSocio= class(TdominioPai)
Private
{ Private declarations }
public
{ Public declarations }
constructor Create;
procedure ProcuraSocio;
end;
	TUMTSocio= class
	Private
	{ Private declarations }
	FNome:STRING;
	FEndereco:STRING;
	Ftelefone:STRING;
	Fcodigo:STRING;
	function R_Nome:STRING;
	procedure W_Nome(umNome:STRING);
	function R_Endereco:STRING;
	procedure W_Endereco(umEndereco:STRING);
	function R_telefone:STRING;
	procedure W_telefone(umtelefone:STRING);
	function R_codigo:STRING;
	procedure W_codigo(umcodigo:STRING);
	public
	{ Public declarations }
	property Nome:STRING read R_Nome write W_Nome;
	property Endereco:STRING read R_Endereco write W_Endereco;
	property telefone:STRING read R_telefone write W_telefone;
	property codigo:STRING read R_codigo write W_codigo;

	end;

Figura 3-8 Exemplo de Classes Tgerente e TumObjeto.

A classe TUmObjeto, fica com os atributos declarados na modelagem da classe, respeitando a sua visibilidade. Se for privado, é criado um atributo nessa seção da classe, com um “F” na frente do nome do atributo, e declarado seu tipo. Se for do tipo public, o atributo é declarado da mesma forma na seção privada da classe, porém são criados métodos para leitura e escrita do atributo, além de uma propriedade com o nome do atributo, que acessa através dos métodos o valor do atributo. A função de leitura do atributo é criada com um “R_” seguido do nome do atributo, já a procedure de escrita, é criada com um “W_” seguido do nome do atributo.

Da classe do tipo gerente, apenas o corpo do método contrutor Create é gerado, os outros métodos que são declarados na modelagem têm apenas suas declarações e seus corpos vazios incluídos no código gerado.

constructor TTSocio.Create;
begin
inherited Create;
with tabela do
begin
TableName := TabelaTSocio;
Open;
end;
end;
	Aponta a tabela para a constante declarada anteriormante que contém o nome do arquivo paradox, e abre a tabela.

procedure TTSocio.ProcuraSocio;
begin
end;
end;
	Método vindo da modelagem, somente é gerada sua declaração.

Figura 3-9 Exemplo de código da classe Tgerente.

Os métodos das classes do tipo TUmObjeto, que devem gerados, são aqueles

que irão dar para as propriedades acesso aos valores dos atributos públicos das classes. A seguir apenas um exemplo apresentado em PINTO e BASTOS (2000) da função de leitura e do procedimento de escrita em um atributo, todos os outros seguem o mesmo estilo.

function TumTSocio.R_Nome:STRING;
begin
  result:=Fnome;
end;
procedure TumTSocio.W_Nome(umNome:STRING);
begin
  FNome:=umNome;
End;
	Retorna o valor de Fnome.
	
	Atribui um valor a Fnome.

Figura 3-10 Função e procedimento para acesso a propriedades.

3.2.3 Interface com o usuário: Classes de interface

A identificação dos objetos de interface facilita a visualização e o reuso dos objetos desse tipo. No desenvolvimento de sistemas comerciais as classes de interface pré-definidas vão sendo colocadas nos sistemas, sem qualquer tipo de modificação, o que compromete a robustez e reutilização destes. Para que estas classes possam ser inseridas nos sistemas, as regras básicas de ***encapsulamento, coesão e herança*** devem ser seguidas.

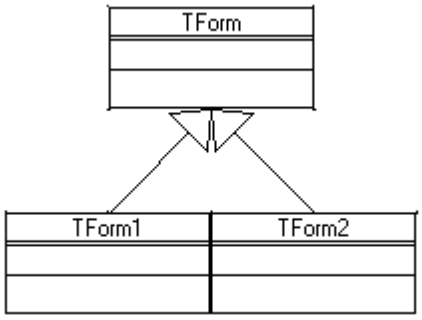

Figura 3-11 Herança entre Classes de Interface

3.2.4 Fluxo de controle do sistema: Classes de controle

Um objeto de controle é um módulo que implementa a lógica de controle de um caso de uso. Cada caso de uso gera um novo objeto de controle. Uma máquina de estado descreve o comportamento dinâmico de um objeto. O objeto é descrito em termos de seus estados e suas reações às mudanças de estado. Este objeto não tem atributos, e apresenta como serviços somente aqueles que retornam resultados de operações solicitados pelo objeto de controle. Assim, o objeto de controle é uma espécie de gerente do caso de uso, indicando as operações e a ordem em que elas devem ser executadas. (SCHMITZ e SILVEIRA, 2000)

As classes de controle possuem, inicialmente, dois serviços, que são: o que inicia o caso, e o de tratar as sinalizações provindas da interface, esses serviços são gerados independentes do modelo. O que inicia o caso, chamado de IniciaCaso, basicamente, cria as classes de interface e as classes de domínio que serão utilizadas, e faz o método “Callback”, da interface, receber o endereço da rotina TrataEvento, que é do tipo procedural (Ver Apêndice C). Dessa maneira a classe de interface, irá delegar seus eventos a classe de controle sem conhecê-la.

```
Procedure TrataEvento (Msg : integer);
Begin
  Case Status of
     Estado1: case Msg of
                        M1:
                        M2:
                End;
     Estado2: case Msg of
                        M1:
                        M2:
                End;
  end
end;
```

Figura 3-12 Estrutura do método TrataEvento.

O método TrataEvento é o que irá tratar as sinalizações provindas da interface. A especificação de TrataEvento permite que cada um dos eventos da interface chame esta rotina com diferentes valores de Msg (parâmetro da rotina), além disto, a variável Status descreve o estado corrente do caso de uso. Esse método conterá uma estrutura de dois "case's" onde, de acorodo com o estado da classe de controle e o evento ocorrido na interface, um conjunto de métodos será realizado. Basicamente, essa estrutura é a implementação dos diagramas de transição de estados contidos no modelo.

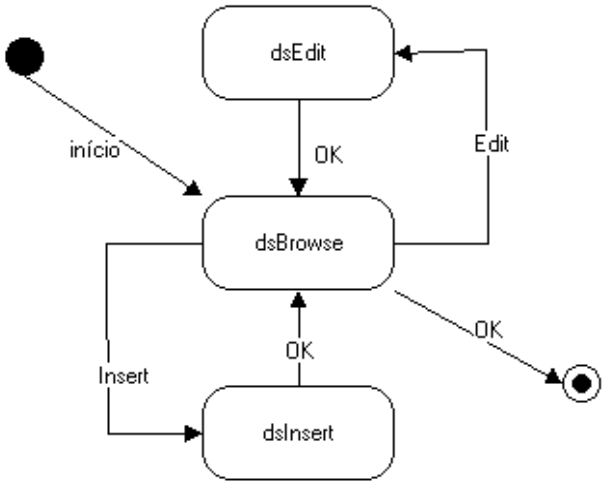

Figura 3-13 DTE de Caso de Uso do Tipo Mantendo

Para os casos de uso do tipo “mantendo”, foi adotado um diagrama de estados genérico, utilizando-se de algumas propriedades existentes nos componentes do ambiente *Delphi*. Os eventos de “edição” e “inserção” nas classes de domínio são disparados automaticamente pelo componente do tipo TDBNavigator.

3.2.5 A recuperação das informações

Para não haver perda de informações entre as iterações, deve ser realizada uma varredura nos arquivos do projeto *Delphi* contendo as informações já geradas e que possuem os códigos adicionados manualmente, para que não seja perdida nenhuma informação.

Para um arquivo de extensão *dpr* ser considerado um arquivo de projeto em *Delphi*, a primeira linha deve conter a palavra-chave ***program*** seguido do nome do projeto. Se esta linha não for encontrada o arquivo é considerado como inválido, sendo encontrada, o que vier após a palavrachave ***program*** é utilizado como nome do projeto (ESPERANÇO *et. al*, 2001).

```
01] program Prototipo;
02]
03] uses
04]   uPrin in 'uPrin.pas' {frmPrin}{,
05]   unit2 in 'unit1.pas',
06]   unit3 in 'unit2.pas',
07]   unit3 in 'unit3.pas';
08]
09]{$R *.RES}
10]
10]begin
10]  Application.Initialize;
10]  Application.CreateForm(TfrmPrincipal, frmPrincipal);
10]  Application.Run;
10]end.
```

Figura 3-14 Estrtura do arquivo de projeto do Delphi.

Após a palavra-chave ***program*** segue a palavra-chave ***uses***, e então uma lista dos módulos usados pelo projeto, esta lista pode apresentar os seguintes formatos:

```
NomeUnit in NomeUnit.pas | NomeUnit in NomeUnit.pas{frmNomeUnit}
```

Onde a primeira *palavra-chave* é o nome que o compilador *Delphi* utiliza para se referenciar ao módulo, o segundo é o nome físico módulo, e o terceiro, se houver, aparece entre chaves e indica o formulário que está associado ao módulo (ESPERANÇO *et. al*, 2001).

3.3 Processo de desenvolvimento usando o Gerador de Programas

A figura 3-15 mostra, de uma forma genérica, o modelo do processo de desenvolvimento de um sistema de informação, utilizando o GP e a MRDS.

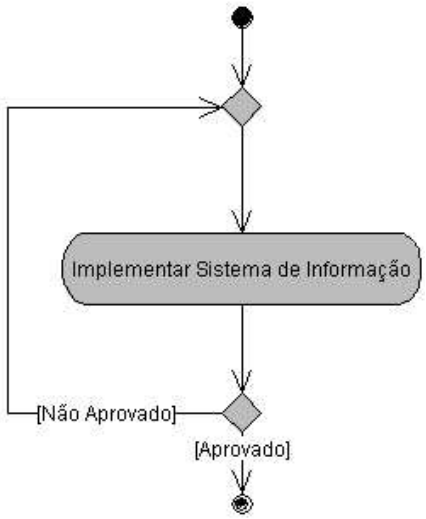

Figura 3-15 Processo da utilização do GP

A atividade “Implementar versão do sistema de informação” pode ser decomposto em sub-atividades, como mostrado na figura 3-16:

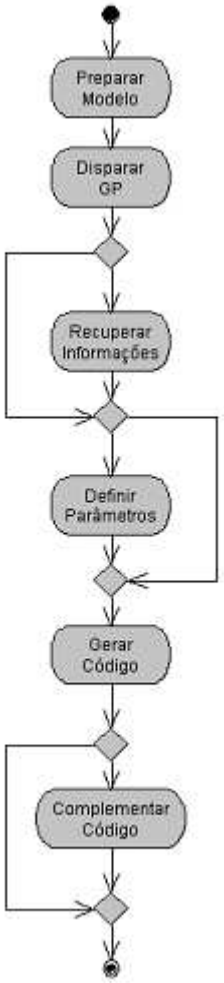

Figura 3-16 Detalhamento do processo Implementar Versão do SI.

A seguir vamos descrever cada uma das atividades mostradas no diagrama da figura 3-16, detalhando cada passo do processo de utilização do GP.

3.3.1 Preparar Modelo no Fast Case

O usuário começa o seu trabalho através do Fast Case, ferramenta desenvolvida por SILVEIRA (1999), preparando os modelos necessários para a geração do código e implementação de uma versão do sistema de informação. Os modelos necessários seguem a MRDS, descrita em SCHMITZ e SILVEIRA (2000), e são eles: o modelo de casos de uso, o modelo de classes e o modelo dinâmico.

O modelo de classes precisa estar estruturado de forma que exista um diagrama contendo somente as classes de domínio do sistema de informação que está sendo desenvolvido e suas associações, essa informação é necessária para que seja possível o mapeamento das classes de domínio para um banco relacional. Cada classe no modelo de classes precisa estar associada a um dos três tipos de estereótipos defindios pela MRDS (interface, controle e domínio), quando não é definido nenhum estereótipo para uma classe, o GP indicará ao usuário, através de um ponto de interrogação na interface que permite ao usuário navegar pelas classes do modelo, que a classe está sem estereótipo associado. No modelo de classes deverão existir, para cada caso de uso concreto, uma e somente uma classe de controle, uma classe de interface e “N” classes de domínio.

O modelo dinâmico, por sua vez, contém dois tipos de diagramas: o diagrama de seqüência e o diagrama de transição de estados.

O usuário deverá fazer um diagrama de sequência para cada caso de uso concreto por ele definido, o diagrama de seqüência deverá ser o “caminho feliz” do caso de uso e deve conter os três tipos de classes (interface, controle e domínios) do caso de uso. No diagrama de seqüência teremos então uma classe de controle, na maioria das vezes uma classe de interface e N classe de domínio.

Para cada classe de controle, existe uma para cada caso de uso concreto, deverá ser feito um diagrama de transição de estados. O diagrama de transição de estados vai representar o ciclo de vida da classe de controle. Esse diagrama conterá as chamadas dos serviços das outras classes (interface e domínio) e teremos nela toda a seqüência da programação. É implementada uma arquitetura centralizada nas classes de controle, sendo possível com isso ter a noção da seqüência do sistema de informação desenvolvido, apenas através das classes de controle.

O usuário deve adicionar na classe de controle, atributos dos tipos das outras classes que ele precisará fazer chamadas a serviços no diagrama de transição de estados, para que a classe de controle possa “enxergar” os serviços dessas outras classes e consiga fazer referência a eles na construção do diagrama de transição de estados.

Inicialmente no diagrama de transição de estados, a classe só terá acesso aos seus próprios serviços.

É importante ressaltar que para cada iteração da utilização do GP, é importante construirmos no mínimo um caso de uso, um diagrama de classes, um diagrama de seqüência e um diagrama de transição de estados, e que o aluno vá adicionando casos de uso, classes, diagramas de seqüências e diagramas de transição de estados incrementalmente, permitindo com que em cada iteração do processo seja gerada uma versão do sistema de informação em desenvolvimento.

3.3.2 Disparar GP

Neste ponto do processo, o GP é acionado e começa a interagir com o usuário, o GP carrega toda a informação existente no repositório do Fast Case e trabalha, a partir daqui, independentemente da ferramenta de modelagem.

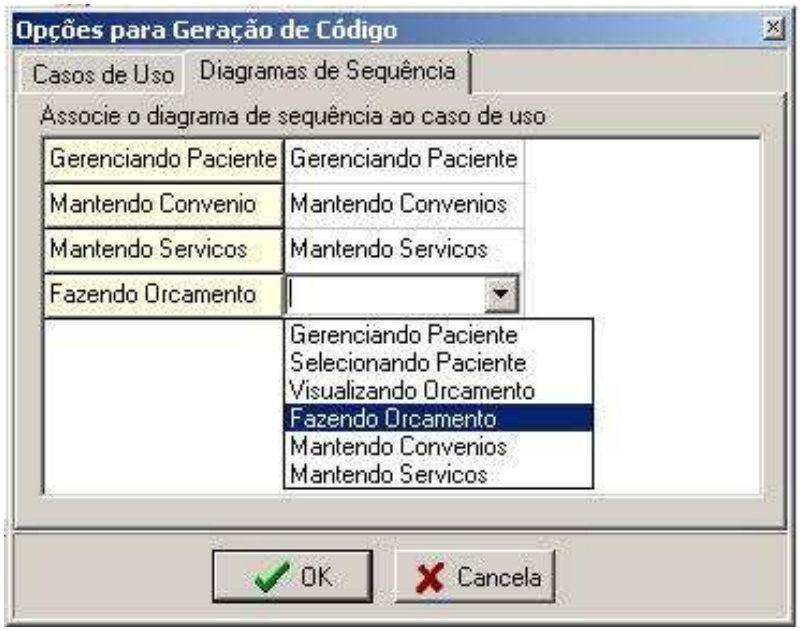

Figura 3-17 Tela onde é feito o relacionamento entre os casos de uso e o diagrama de seqüência.

O usuário deverá informar a correspondência entre os diagramas de sequência e os diagramas de caso de uso, essa dependência é necessária para que o GP faça a

ligação entre todos os diagramas, pois não existe nada no Fast Case que associe um diagrama de seqüência a um caso de uso, pois nada impede pela definição da UML que um caso de uso possa ter mais de um diagrama de seqüência, mas para que o GP trabalhe corretamente essa associação deve existir.

Figura 3-18 Tela onde éinformado os casos de uso do tipo mantendo.

O GP pede também que sejam informados os casos de uso que são do tipo “mantendo”, pois esse é um conceito introduzido pelo GP e o Fast Case não especifica.

3.3.3 Recuperar Informações

Após a preparação dos modelos, o usuário poderá então recuperar as informações por ele já acrescentadas manualmente ao código gerado em uma iteração anterior do processo.

Figura 3-19 Menu recupera informações.

A atividade de recuperar informações, tem como objetivo dar maior produtividade e fechar o ciclo de trabalho para o usuário. Permite que seja realizado o desenvolvimento incremental do sistema de informação, gerando assim, versões funcionais do mesmo. Limitou-se, no GP, a recuperar informações apenas das implementações dos métodos das classes já defindidos no modelo do Fast Case. Apenas os "corpos" dos métodos e as interfaces existentes são recuperados criando uma disciplina no desenvolvimento do sistema de informação para que quaisquer classes, métodos e atributos sejam definidos e acrescentados no modelo, pois essa informação não é recuperada pelo GP.

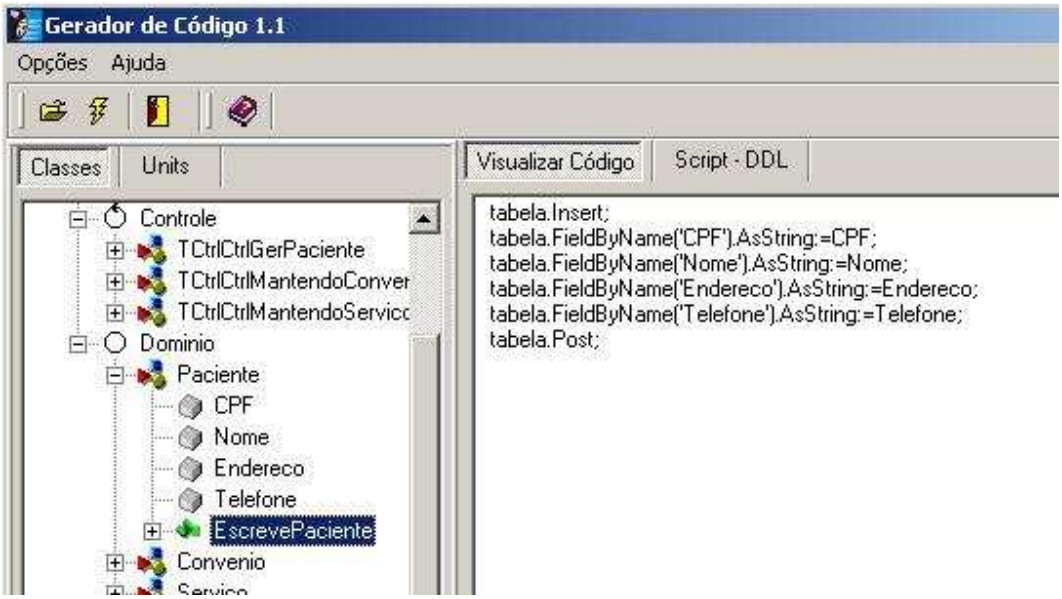

Figura 3-20 Código adicionado manualmente e recuperado para nova geração.

Quando é realizada a recuperação das informações, o GP disponibiliza a visualização das informações que foram recuperadas nos respectivos métodos. Basta marcarmos o método na estrutura de visualização das classes disponibilizada pelo GP, para que a informação recuperada seja mostrada no painel ao lado direito, na aba "Visualizar Código".

3.3.4 Definir Parâmetros no Gerador de Programas

É disponibilizada pelo GP, uma estrutura que permite a visualização e a navegação entre as classes modeladas e/ou arquivos (units) da versão do sistema de informação que será gerado, onde podem ser vistas todas as propriedades dessas classes e/ou das units, de acordo como que foi definido no modelo. O usuário tem também a sua disposição uma série de parâmetros que podem ser definidos de acordo com suas necessidades. Esses parâmetros são: assistente de interface, opções das unidades e opções do domínio.

Figura 3-21 Menu assistente de interface.

O assistente de interface é dividido em duas opções: Importação da maquete (arquivo dfm, que são arquivos que descrevem a interface com o usuário no ambiente *Delphi*) e um assistente de criação de interfaces. A opção de importação da maquete permite que seja importada para uma classe de interface, que deverá estar marcada na estrutura de classes dsponibilizada pelo GP, a maquete que foi construída na fase de análise de requisitos da MRDS, permitindo um aproveitamento e evitando um retrabalho na construção das interfaces.

Figura 3-22 O assistente de criação de interface (GUI).

Existe também um assistente de criação de interfaces que permite que seja criada uma interface simples para a classe escolhida, que também deverá estar marcada na estrutura de classes disponibilizada pelo GP.

Figura 3-23 Opções de Units.

As opções das unidades (units) permitem ao usuário ter a liberdade de escolher com que o GP gere somente alguns arquivos (não gerar todos os arquivos do sistema de informação), além de permitir que possa ser escolhido o formato em que os arquivos "dfm" serão gerados. Estes arquivos podem estar no formato binário ou texto, essa característica se deve ao fato de que versões do ambiente *Delphi* anteriores a versão "5.0" utilizarem arquivos de extensão "dfm", que definem a interface gráfica com o usuário (GUI), no formato binário.

Figura 3-24 Opções de domínio.

As opções de domínio permitem ao usuário escolher se as tabelas em Paradox serão geradas, isto é, se o mapeamento das classes do modelo OO definido, será mapeado para o modelo de tabelas relacionais através de arquivos Paradox.

Permite também que o usuário escolha se o gerador irá gerar um arquivo contendo o "script", fruto do mapeamento OO-relacional realizado pelo GP, podendo ser escolhido o tipo de banco de dados relacional a ser usado (Oracle, Access, SQL Server ou Paradox) para a geração da DDL. Esse arquivo poderá ser executado no banco de dados correspondente para que seja criada a base de dados para o sistema de informação. O GP disponibiliza a visualização da DDL antes de sua efetiva geração, na

aba “Script (DDL)” do lado direito da interface gráfica do GP.

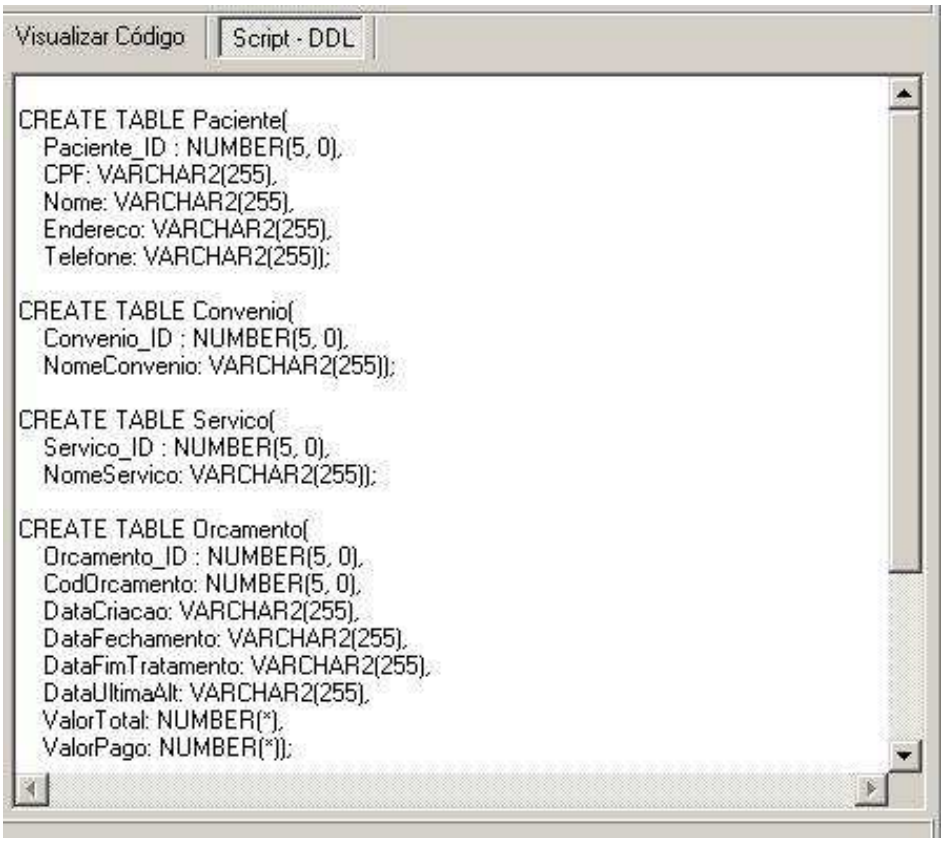

Figura 3-25 Visualização do Script de criação do banco de dados.

3.3.5 Gerar Código

A qualquer momento a partir do acionamento do GP, o usuário está apto a disparar a geração dos arquivos que vão compor o código fonte do sistema de informação. O GP solicita que o usuário escolha a pasta e o nome do arquivo do projeto *Delphi* que será criado.

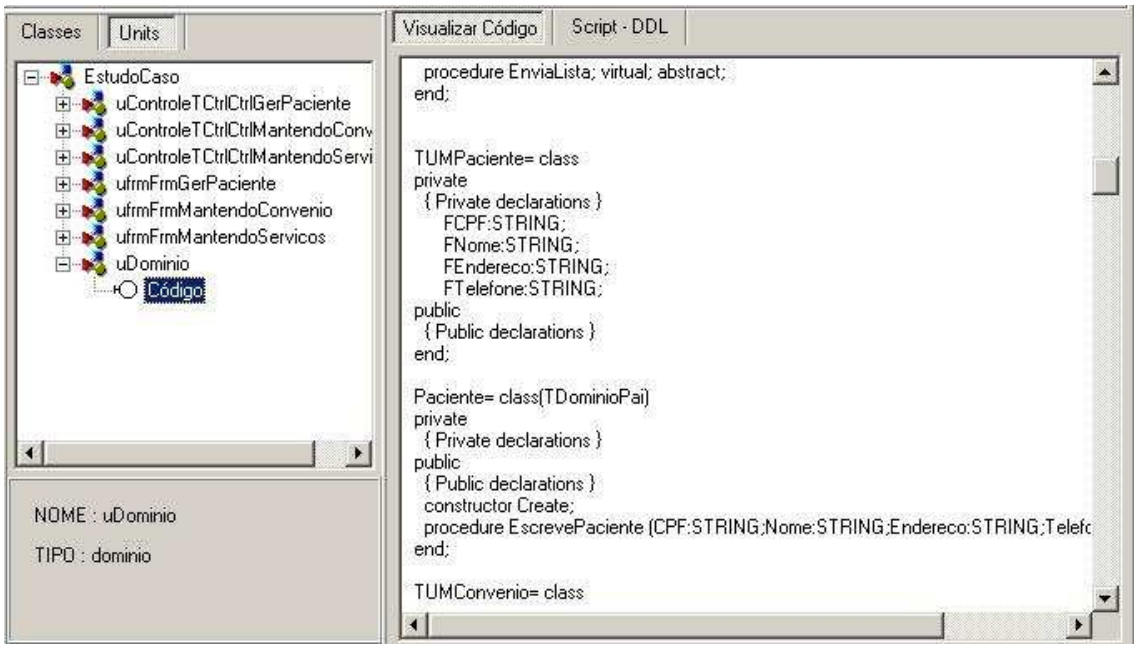

Figura 3-26 Visualização do código que será gerado.

O GP disponibiliza a visualização do código antes de ser gerado, permitindo ao usuário visualizar aquilo que efetivamente será gerado nos arquivos que farão parte do sistema de informação em implementação.

O código em *Delphi* é gerado respeitando uma divisão dos arquivos em pastas da seguinte maneira:

Pasta	Conteúdo
Principal (pasta escolhida para geração)	Interface Principal (.dfm e .pas), o projeto (.dpr) e a unit uConstantes.pas
Classes Abstratas	UfrmManutencaoGeral.pas
Controle Caso	Arquivos contendo as classes de controle (.pas)
Data Modulo	Arquivos do Domínio (.pas e .dfm)
Dcu	Arquivos do projeto após compilação (.dcu)
Dcu/Base	Tabelas em formato paradox e/ou scripts de criação do banco (ddl) das classes de domínio.
Forms	Arquivos de Interface com o usuário do sistema.
Backup	Pasta que irá conter toda a estrutura e os arquivos da versão anterior do SI gerada pelo GP.

Tabela 3-1 Estrutura das pastas e arquivos gerados.

A geração dos arquivos é realizada percorrendo e traduzindo as informações existentes no modelo do GP criado em memória. Este modelo é alimentado tanto por informações vindas do modelo do sistema de informação a ser implementado, definido no Fast Case, como de informações recuperadas de arquivos que foram resultados de gerações de iterações anteriores do GP.

3.3.6 Complementar Código

Após a geração dos arquivos do sistema de informação, o usuário poderá abrir o projeto gerado, no ambiente *Delphi*, compilar e executar o sistema. Em seguida, poderá complementar parte do código fonte manualmente para que o sistema atenda o que o usuário espera do sistema. A complementação do código é realizada sem intervenção do

GP, e espera-se que nesta fase, sejam alterados e implementados somente os corpos dos serviços já existentes e definidos no modelo UML. Essa complementação realizada de forma disciplinada, apenas dos métodos já existentes no modelo, poderá ser recuperada em uma iteração posterior, possibilitando que não sejam perdidas informações já codificadas pelo usuário. O sistema de informação sendo aprovado, é finalizado então o seu processo de impementação, se não for aprovado, faltando ainda algumas iterações para completá-lo, todo o processo a partir da preparação do modelo poderá e deverá ser repetido, permitindo ao usuário uma maior produtividade e um desenvolvimento iterativo do sistema em questão, sem a perda de informações.

3.4 Detalhamento interno do Gerador de Programas

O Gerador de Programas foi desenvolvido em linguagem Object Pascal, a mesma linguagem para qual ele gera código fonte e realiza a engenharia reversa. A escolha desta linguagem se deve a dois fatores principais: (1) os alunos, que são os principais usuários do GP, têm como base na sua formação na UFRJ a linguagem Object Pascal e (2) a ferramenta FAST CASE já ter sido desenvolvida em *Delphi*, e utilizar alguns dos seus tipos e classes na definição dos seus modelos. Mas existe a possibilidade que as mesmas soluções e os mesmos métodos utilizados no GP possam ser implementados em outras linguagens de programação e que o GP possa também gerar código fonte em outras linguagens de programação.

Das atividades mostradas na figura 3-16 existem três que podem ser descritas através de uma visão interna de funcionamento do GP, são elas: “Disparar GP”, “Gerar Código” e “Recuperar Informações”.

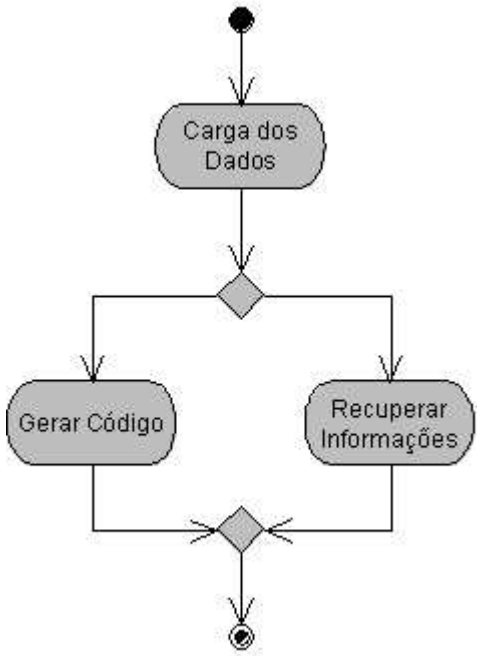

Figura 3-27 Processos internos do GP.

3.4.1 Disparar o GP

O primeiro passo realizado pelo gerador é a leitura do repositório da ferramenta Fast Case, percorrendo todos os seus arquivos de armazenamento, e carregando no modelo do GP as informações necessárias e existentes na modelagem do sistema de informação escolhido. O modelo do GP é uma extensão do modelo da ferramenta Fast Case.

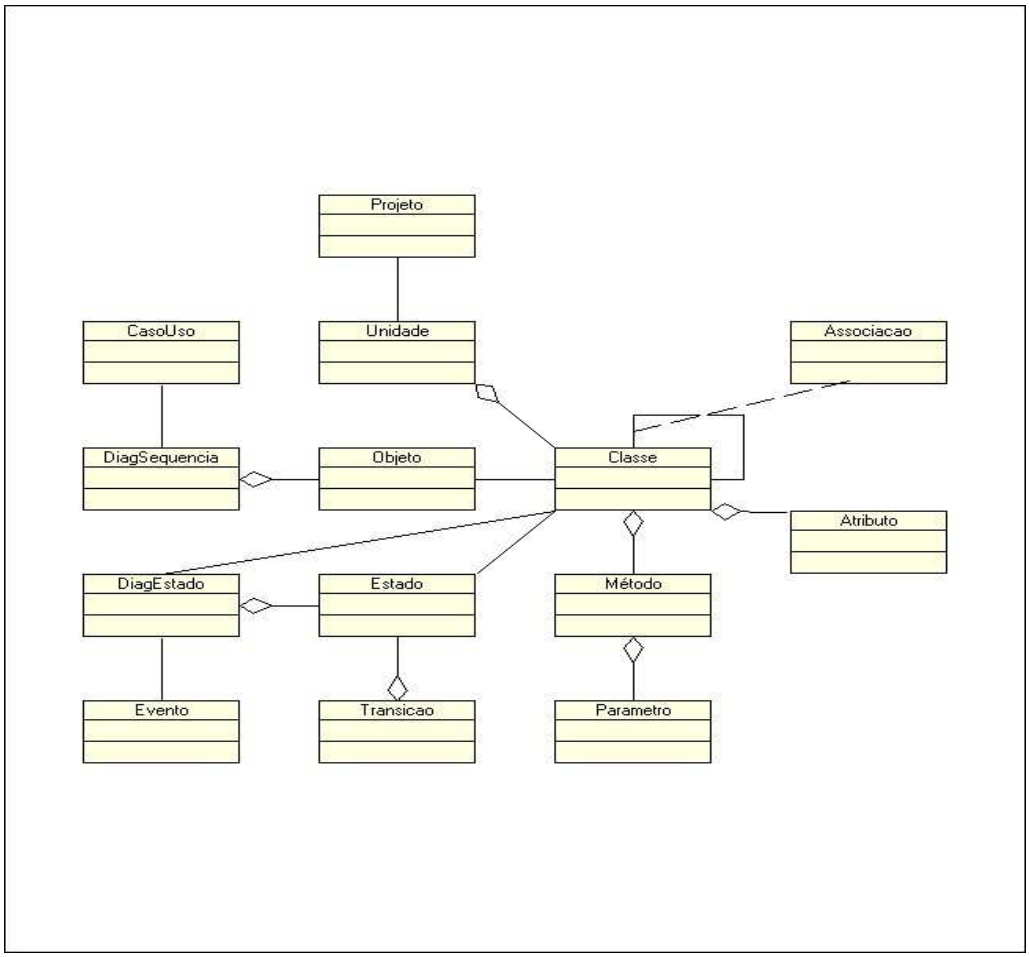

Figura 3-28 Modelo de classes de domínio do GP

No momento da carga dos dados, o GP realiza a geração do código de alguns arquivos permitindo que o usuário veja o que deverá ser gerado utilizando os parâmetros padrões definidos pelo GP. Essa visualização pode ser observada na aba "Visualização de Código" no painel ao lado direito do GP.

3.4.2 A atividade Gerar Código

A atividade "Gerar Código" pode ser decomposta nas sub-atividades mostradas na figura 3-29.

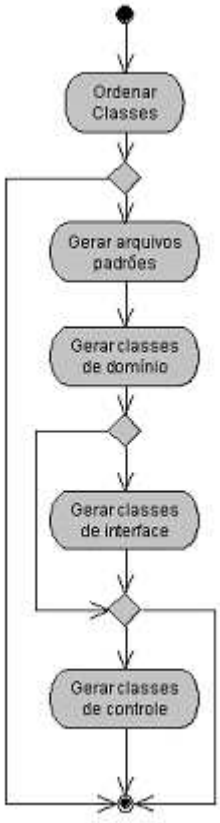

Figura 3-29 Sub-atividades da atividade Gerar código.

3.4.2.1 Ordenar Classes de domínio

É realizada uma ordenação topológica[5] das classes para a verificação da existência de ciclos nas associações de herança do modelo de classes de domínio, o que indicaria uma inconsistência no modelo de classes. Existindo ciclos é indicada ao usuário a inconsistência de sua modelagem.

[5] Ver o apêndice para mais detalhes sobre a ordenação topológica.

3.4.2.2 Gerar Arquivos Padrões

Essa atividade pode ser decomposta nas sub-atividades descritas na figura 3-30:

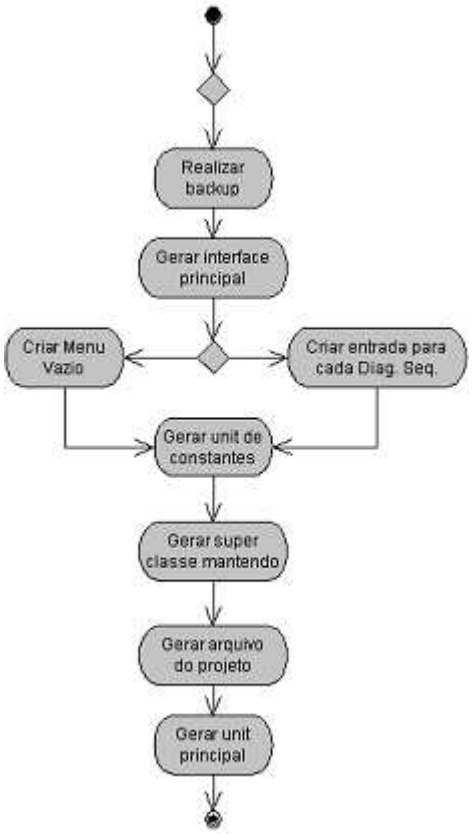

Figura 3-30 Gerar arquivos padrões

O GP verifica a existência anterior de algum projeto, isso ocorre quando não é a primeira geração de código para esse projeto na mesma pasta. Existindo um projeto anterior, é realizado um backup, criando uma pasta com o mesmo nome (“backup”) e o gerador percorre todos os subdiretórios do projeto e salva os arquivos na mesma estrutura de diretórios utilizados para geração, só que dentro da pasta “backup”.

A interface principal é a próxima a ser gerada. Para este fim, é criado um formulário (.dfm) e uma unit (.pas), apenas um esqueleto básico é gerado, suficiente para a execução do programa. Essa interface contém apenas um “menu” com as opções de “Funções”, “Mantendo”, “Sobre” e “Sair”. Nesse passo é percorrido todo o modelo

procurando-se por diagramas de seqüência, pois são esses diagramas que nos indicam a existência da parte dinâmica do projeto. Para cada diagrama de seqüência existente é criada uma entrada no menu da interface principal. A não existência de diagramas de seqüência leva a geração de uma interface principal com um menu vazio, contendo apenas as entradas básicas já mencionadas.

A unit contendo as constantes é então gerada logo em seguida. Essa unit conterá, dentre outras constantes, o tipo procedural (ver Apêndice C) que define o funcionamento das chamadas dos serviços quando ocorre um evento em alguma interface. (“TEvento = procedure (Msg: integer) of object;”).

Cada caso de uso especificado como do tipo “Mantendo”, possui basicamente o mesmo comportamento e sua interface tem a mesma aparência, logo é gerada uma unit de interface (.pas e .dfm) contendo uma classe abstrata, de onde todas as classes de interface do tipo “mantendo” irão herdar suas propriedades e funcionalidades.

O arquivo principal do projeto (.dpr) é gerado a seguir. Esse arquivo contém tanto a lista das units que compõem o projeto e seus respectivos caminhos, assim como a chamada para a interface principal que iniciará a execução do sistema de informação em implementação.

3.4.2.3 Gerar as classes de domínio

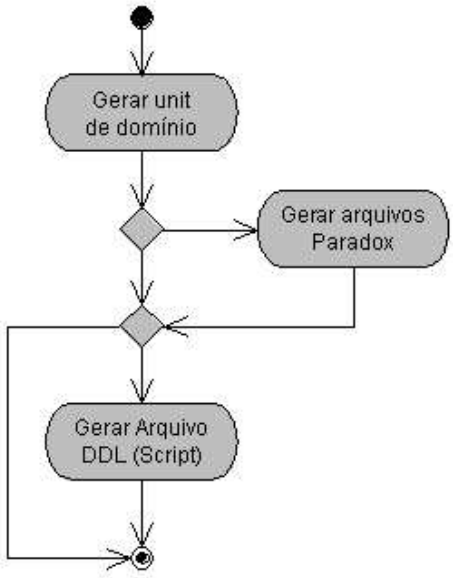

Figura 3-31 Gerar classes domínio

Todas as classes de domínio são herdeiras de uma classe genérica chamada de TDominioPai onde existem alguns métodos e atributos comuns a todos os domínios. Em seguida, é verificado o tipo do banco de dados escolhido, sendo o tipo escolhido o paradox (tipo nativo do Delphi), o GP cria para cada classe de domínio além da sua definição de classe, um arquivo (“.db”) correspondente a sua tabela física, através de um mapeamento OO-Relacional que é realizado.

O corpo do método Create da classe TDMDominio é criado automaticamente, fazendo com que o database aponte para o local (“dcu/base”) onde foram criadas as tabelas Paradox, fazendo com isso que essas possam ser acessadas pelo sistema.

```
procedure TdmDominio.dmDominioCreate(Sender: TObject);
begin
  Database.Params[0]:='PATH='+ExtractFilePath(Application.ExeName)+'DCU\BASE';
end;
```

Figura 3-32 Método Create do DataModule.

Sendo escolhida a opção para geração do arquivo de script (DDL), o GP gerará esse arquivo, respeitando a sintaxe da opção de banco de dados escolhida.

Por último são gerados os scripts para geração da base de dados (SQL- DDL) tomando com base o conteúdo do diagrama de classes e as classes do tipo “domínio”, que serão as classes que precisarão de persistência. Os scripts são gerados de acordo com opção de banco de dados que foi escolhida (Paradox, Access, SQL Server ou Oracle).

3.4.2.4 Gerar as classes de interface

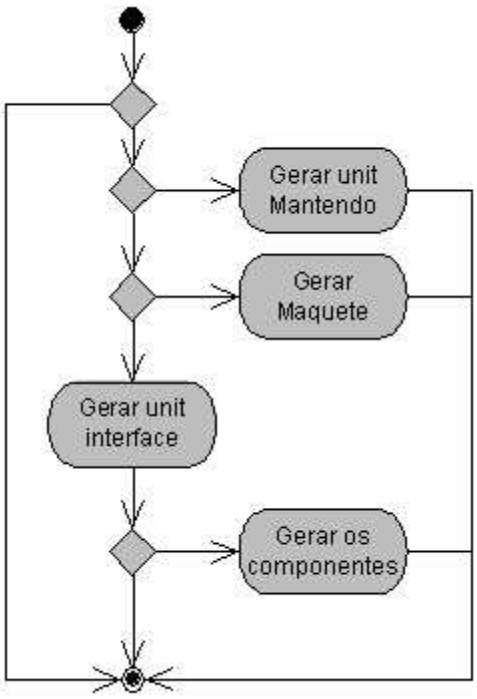

Figura 3-33 Gerar classes de interface

Podem ser gerados três tipos básicos de interfaces (PINTO, SILVEIRA e SCHMITZ, 2000):

A primeira delas é a interface principal do sistema que é filha, como toda classe de interface, da classe TForm. Esta interface contém um menu é tem itens de menu para cada diagrama de seqüência definido no modelo Fast Case.

A segunda é uma interface básica (Tipo b), herda também de TForm, para onde cada classe de interface é mapeada. Cada uma delas é acrescida de um botão do tipo TBitBtn.

A terceira interface (Tipo c) é aquela em que são mapeados os pacotes de casos de uso do tipo “mantendo”. Essa interface herda de uma classe padrão construída e que utiliza a facilidade existente nos componentes do ambiente *Delphi*, que permite verificar o estado de uma tabela. Essa classe pai foi chamada de TFrmManutencaoGeral que por sua vez herda de um TForm, possui ainda um TDBNavigator, um TDBGrid e um TBitBtn.

As duas interfaces citadas acima (Tipo b e Tipo c), possuem um método chamado de Callback, do tipo procedural (ver Apêndice C) que receberá o método da classe de controle que irá tratar os eventos da interface. Desta forma a classe de interface não precisará “conhecer” em tempo de compilação quem trata seus eventos, possibilitando um máximo de desacoplamento entre a interface e as outras classes do sistema, permitindo que ela seja reutilizada em vários casos de uso (PINTO e BASTOS, 2000), além de facilitar uma futura troca de interface.

A seguir o GP verifica a existência de classes com o estereótipo “interface”. Para cada classe de “interface” é verificado então, percorrendo-se a estrutura que armazena os diagramas de seqüência, se o diagrama de seqüência que contém essa classe é de um caso padrão do tipo “mantendo”.

Se sim, é gerada uma unit que conterá a classe do tipo “interface”, como filha da classe Pai dos casos mantendo já gerada. Se o diagrama de seqüência não é de um tipo “mantendo”, então é verificado se o usuário importou alguma maquete, associando a essa classe de interface.

Se existe uma maquete associada, é gerada a interface, somente a parte visual (.dfm) é originada da maquete. Se não existe maquete associada, então é gerada uma unit contendo a classe de interface, com seus atributos e serviços. O gerador verifica também nesse ponto se a classe de interface foi construída através do assistente

existente no próprio gerador, se sim, os componentes escolhidos no assistente são inseridos no arquivo ".dfm" gerando a interface como definida pelo usuário no assistente.

3.4.2.5 Gerar as classes de controle

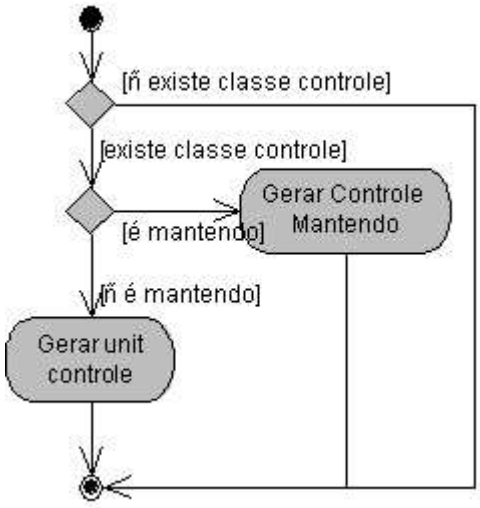

Figura 3-34 Gerar classes de controle

Nesta etapa o GP percorre a estrutura da lista das classes, verificando a existência de classes com o estereótipo "controle". Para cada classe de "controle" é verificado então, percorrendo-se a estrutura que armazena os diagramas de seqüência, se o diagrama de seqüência que contém essa classe é de um caso padrão do tipo "mantendo".

Se sim, é gerada uma unit que conterá a classe de controle, porém com os serviços das classes já definidos como padrão para os casos "mantendo". Se não, a unit gerada conterá a classe de controle específica com seus atributos e métodos.

3.4.3 A atividade Recuperar Informações

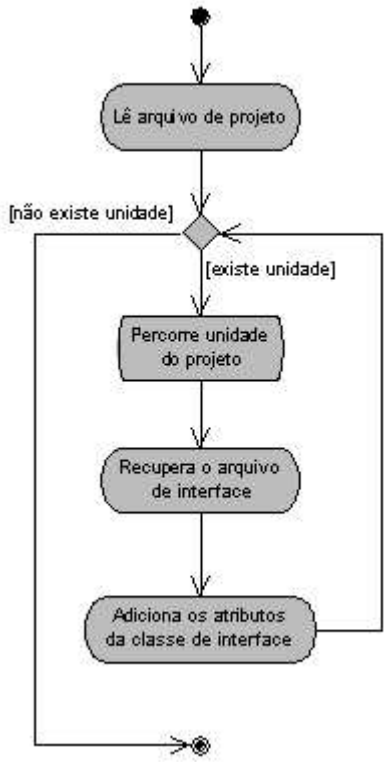

Figura 3-35 Sub-atividades da atividade Recuperar Informações

O GP lê o arquivo da versão anterior do projeto do sistema de informação em desenvolvimento e para cada unidade (unit) do projeto o GP verifica se existe um arquivo de interface associada a unit, indicando que a classe correspondente àquela unidade é uma classe de interface. Existindo, o GP carrega a maquete permitindo que, na geração, seja criada a mesma interface já definida na versão anterior.

Em seguida, o GP percorre os arquivos do programa, localizando a seção “ïmplementation”, e verificando cada método definido, recuperando todo o código para cada método existente no arquivo e que exista sua definição no modelo.

4 Avaliação da ferramenta

4.1 Evolução da ferramenta

O GP vem sendo utilizado pelos alunos da disciplina Modelagem de Sistemas de Informação II da graduação em Informática da UFRJ desde sua primeira versão, onde foi possível através das observações e contribuições dos alunos realizar melhorias, permitindo uma evolução incremental à ferramenta, agregando funcionalidades para alcançar o principal objetivo de auxiliar o aluno no desenvolvimento iterativo de sistemas, possibilitando ao mesmo um melhor entendimento de metodologias e de arquitetura de software.

O GP teve sua primeira versão contendo apenas o módulo de geração de código, sem nenhuma customização por parte do usuário, era bem simples com relação a interface gráfica (GUI), onde continha apenas uma barra de progressão indicando a geração dos arquivos de código fonte com extensão pas, dpr e dfm, além da geração de arquivos de persistência em Paradox. Essa primeira versão não tinha nenhuma interação com o usuário (aluno), e foi submetida ao XIV Simpósio Brasileiro em Engenharia de Software realizado na Paraíba, onde obteve a premiação de melhor ferramenta do Simpósio (PINTO, SILVEIRA e SCHMITZ, 2000).

Após essa primeira versão foi observada a necessidade de um melhor tratamento da parte de persistência, que havia sido contemplada bem superficialmente na primeira versão, logo foi desenvolvida uma segunda versão, acrescentando ao GP a funcionalidade de geração de scripts, com um enfoque maior e mais detalhado no mapeamento OO-Relacional. Foram abordados os bancos de dados Access, SQL Server e Oracle além do Paradox já contemplado na primeira versão.

Uma terceira versão foi desenvolvida, visando uma maior interação com o usuário. Foi criada a possibiliade de ser importada a maquete (interface gráfica), possibilitando assim maior produtividade e não desperdiçando trabalho realizado nas fases iniciais do MRDS, que prevê a construção de uma maquete na fase de requisitos.

Foi criado um assistente de criação de interfaces, permitindo a criação e geração de interfaces simples a partir do GP.

Após essas três versões, ainda persistia o problema de reaproveitamento de código já gerado anteriormente pelo GP, pois após cada nova geração todo o código era re-gerado, podendo causar perda de informação se o usuário já estivesse acrescentado código manualmente. Com isso uma quarta versão foi desenvolvida, acrescentado-se um módulo de recuperção de código, através de engenharia reversa, permitindo assim com que o ciclo fosse fechado e não houvesse perda de informação dentro do que era esperado. Como o objetivo não é especificamente recuperar código, foi levada em conta apenas recuperação de código dos corpos dos métodos, pois esta informação não está contemplada no GP nem no FAST CASE, utilizado em conjunto. Essa recuperação de código foi desenvolvida com o cuidado de não permitir que o aluno incluisse métodos, atributos, classes e outras informações no código, que não fosse as implementações dos métodos já definidos no modelo UML e quisesse recuperar essas informações para o modelo, pois esse não é o objetivo do GP e desta funcionalidade, e faria o aluno trabalhar de maneira inversa da que realmente deve ser trabalhada, que é do modelo para a implementação.

Versões do GP	**Funcionalidades Adicionadas**
Primeira versão (1º. Lugar no XIV SBES)	Geração de código e banco Paradox.
Segunda Versão	- Mapeamento OO-Relacional - opção para geração de DDL's em outros bancos de dados
Terceira Versão	- Criada uma interface para interação com o usuário. - Opção de importar maquete - Assistente de interface
Quarta Versão (Atual)	- Recuperação de informação (Reversa)

Tabela 4-1 Resumo do histórico das versões do GP

A tabela anterior resume as versões construídas do GP, mostrando em cada uma delas as características principais que foram adicionadas à ferramenta.

4.2 Estudos de casos

Os alunos de graduação do curso de informática da UFRJ realizaram vários casos e exemplos através das várias versões do GP, possibilitando a evolução e correção de problemas. O GP foi utilizado na disciplina de graduação “Modelagem de Sistemas de Informação II” do departamento de informática da UFRJ, em 6 períodos seguidos, dos anos de 2000 ao ano de 2002, num total de 28 projetos. Selecionamos alguns dos últimos trabalhos desenvolvidos por esses alunos e estão documentados no apêndice, sendo possível verificar a eficiência do GP e do processo incremental. Foi realizado também um estudo de caso adicional para observar as dificuldades encontradas pelos alunos na utilizaçao do GP, nos servindo também como base para correções e verificações de falhas no processo de utilização da ferramenta.

Para esse estudo de caso adicional foi escolhido um modelo já implementado pelos alunos para efetivo desenvolvimento utilizando o GP e a MRDS de forma incremental, sendo possível verificar os pontos de melhorias para que esses sejam implementados e/ou documentados para trabalhos futuros.

4.3 Um exemplo com a versão atual

Visando confirmar e verificar a eficiência do GP e do processo incremental que ele sugere, foi desenvolvido, parcialmente, um exemplo para que fosse documentada a evolução do processo de desenvolvimento de um sistema de informação através do GP.

O sistema de informação que foi implementado foi o de uma clínica odontológica, onde a idéia do sistema é controlar uma clínica odontológica, quanto aos seus pacientes, os pagamentos, os atendimentos, as consultas e os convênios existentes. Pela definição original do sistema, foram definimos 12 casos de uso, sendo 2 do tipo mantendo, 10 diagramas de seqüências, e 8 diagramas de transição de estados. Nem

todos os casos foram implementados, pois para o objetivo aqui proposto, observou-se que a partir de um certo momento, passamos a ter uma repetição de passos que não valeria a pena ficar descrevendo.

A seguir apresentamos as iterações que foram realizadas e o que foi feito e definido em cada iteração do processo incremental no desenvolvimento desse sistema.

4.3.1 Primeira iteração

No Fast Case, definimos o caso de uso “Gerenciando Pacientes”, que é um caso que usa um outro caso de uso abstrato “Selecionando Paciente”. Definimos também o caso de uso do tipo mantendo “Mantendo Convênio”.

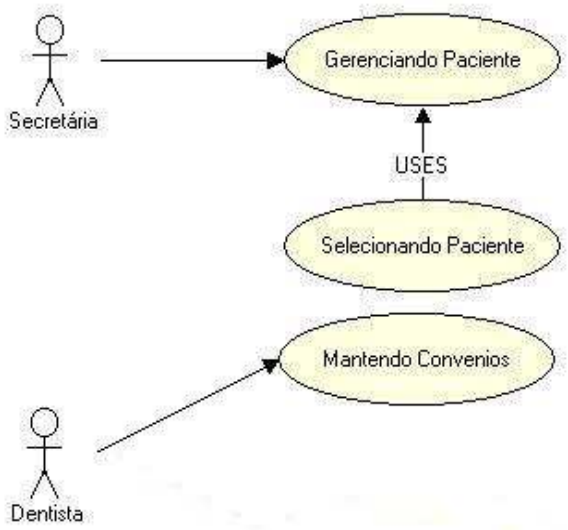

Figura 4-1 Casos de uso da primeira iteração.

Criamos um diagrama contendo as classes de domínio identificadas até o momento, um diagrama de classes para as classes de interface e um diagrama de classes para as classes de controle.

Em seguida criamos dois diagramas de sequência, um para o caso de uso do tipo “mantendo” e um para o caso “Gerenciando Paciente”, obedecendo a arquitetura proposta.

Para cada classe de controle existente, até o momento existiam duas classes de controle, criamos os dois diagramas de transição de estados.

Figura 4-2 Diagrama de transição de estados da classe de controle do caso gerenciando paciente.

Com os diagramas prontos, disparamos o GP, escolhemos para criar o arquivo contendo o "script" de criação do banco de dados e para criar os arquivos em Paradox. Importamos a maquete existente para a classe de interface do caso de uso "Gerenciando Paciente". Em seguida acionamos a opção de geração de código no GP. O projeto é então gerado, aberto no ambiente Delphi e compilado sem problemas.

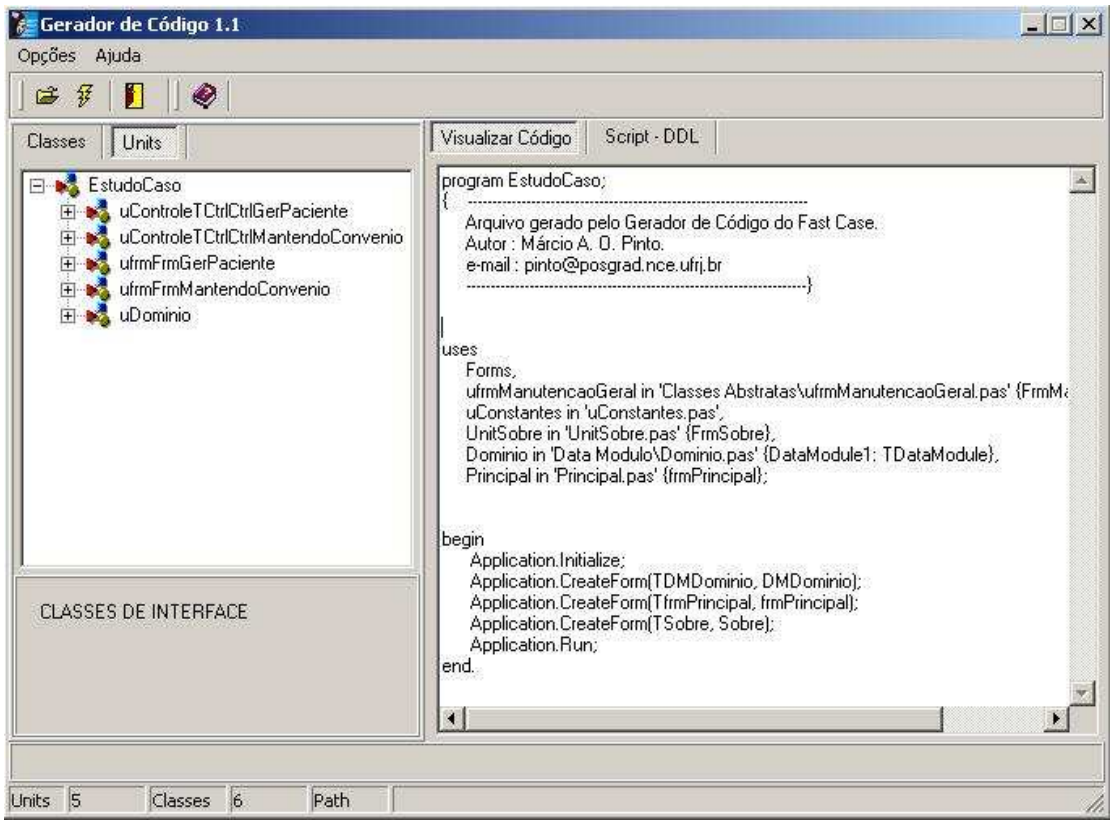

Figura 4-3 O GP pronto para a primeira geração.

Agora complementamos o código fonte gerado, implementando os métodos que foram definidos no modelo para que possamos ter uma versão fuincional do sistema de informação em questão. Os casos de uso do tipo “mantendo” estão totalmente funcionais.

Com isso temos, no final da primeira iteração, a primeira versão do sistema de informação como esperado.

4.3.2 Segunda iteração

Na segunda iteração do desenvolvimento do sistema utilizando o GP, voltamos ao Fast Case, e definimos mais um caso de uso, “Fazendo Orçamento”, que tinha uma associação de <include> com o caso “Visualizando Orçamento”, que também foi adicionado. Adicionamos também o caso de uso do tipo “mantendo”, chamado de “Mantendo Serviços”.

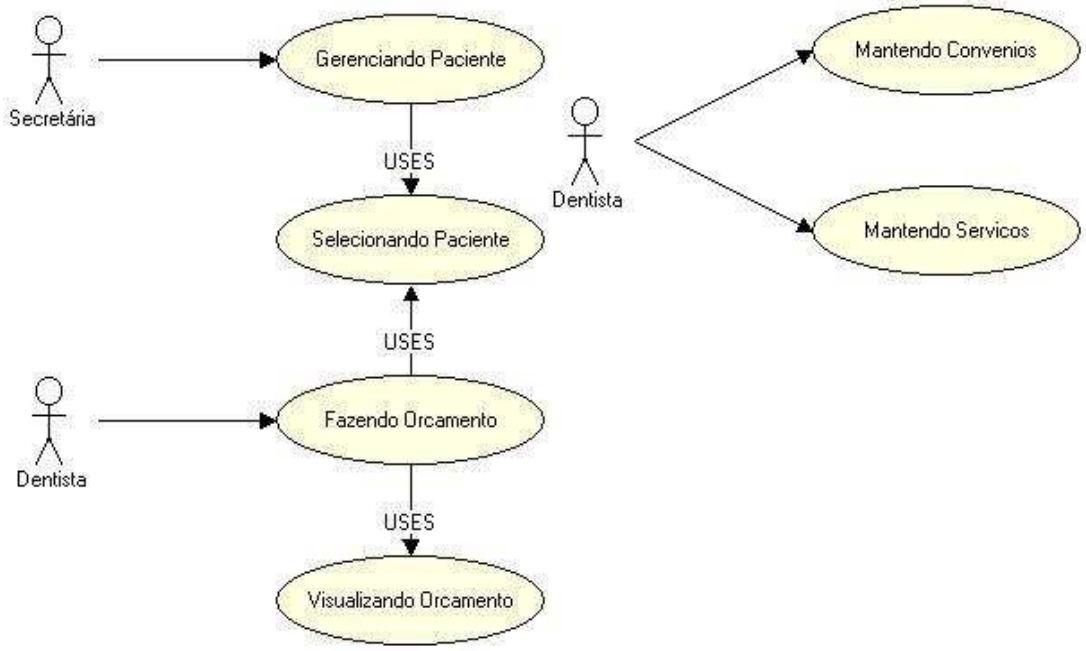

Figura 4-4 Casos de uso após a segunda iteração.

No diagrama de classes adicionamos as classes referentes ao caso de uso, classe de interface, classe de controle e a classe de domínio.

Mais dois diagramas de seqüência são criados, referente aos casos de uso adicionados, e dois diagramas de transição de estados são criados para as classes de controles dos novos casos de uso.

Acionamos o GP, mas agora como é a segunda iteração do processo de desenvolvimento, precisamos disparar o processo de “Recuperação de informações”. Escolhemos como entrada, o arquivo “dpr” gerado na iteração anterior, e que faz parte da primeira versão do sistema de informação. Observamos que os métodos implementados para o caso de uso “Gerenciando Paciente” que foi codificado na primeira iteração são todos recuperados, assim como as interfaces correspondentes. Importamos a maquete referente ao caso adicionado “Alterando Orçamento” e geramos o código para a segunda versão do sistema de informação.

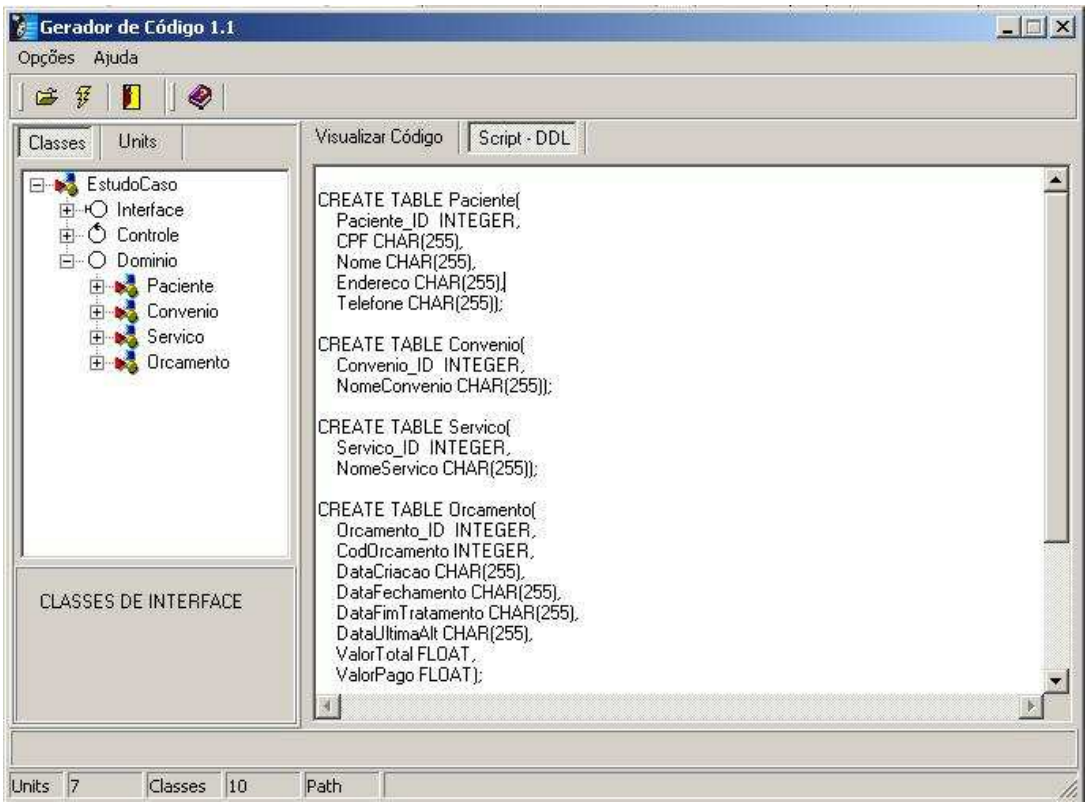

Figura 4-5 O GP pronto para a segunda geração.

Agora complementamos o código fonte gerado, implementando os métodos do novo caso de uso e observamos que os métodos iimplementados na primeira versão foram todos recuperados e estão com na primeira versão, não ocorrendo perda de informação.

Com isso temos agora, uma segunda versão do sistema de informação, com o acréscimo de mais um caso de uso em relação à primeira versão do sistema de informação.

4.3.3 Terceira iteração

Para a terceira iteração, vamos ao Fast Case, adicionamos mais um novo caso de uso, dessa vez o caso de uso “Gerenciando Ficha Clínica”.

Novamente repetimos os passos já conhecidos da MRDS para definir todos os

modelos necessários para o GP. Criamos as classes de interface e de controle referente ao caso de uso adicionado. Agora, no entanto, não foi adicionada nenhuma classe de domínio, pois as classes de domínio utilizadas pelo novo caso já existiam na modelagem. Adicionamos apenas novos serviços na classe referente ao novo caso de uso.

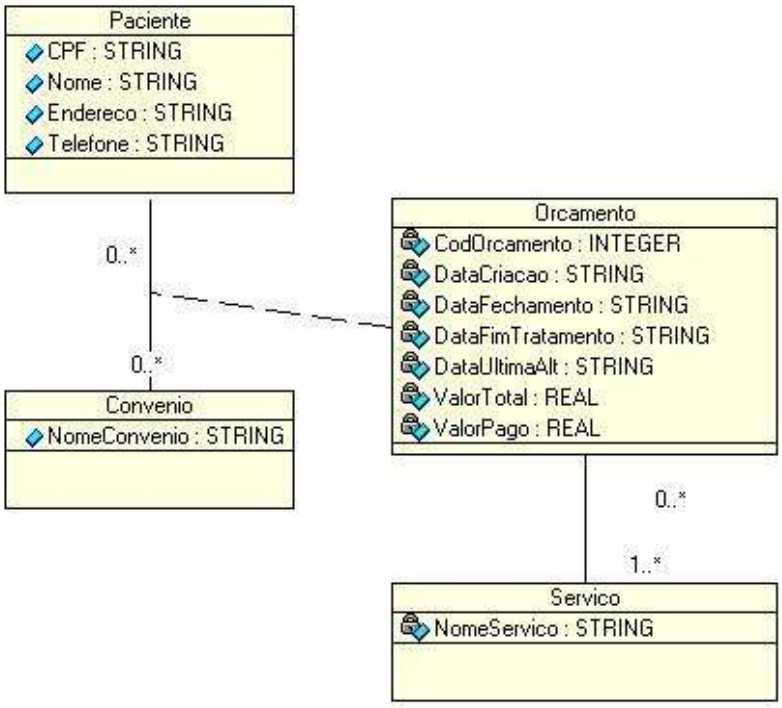

Figura 4-6 O diagrama de classes do domínio.

Criamos então o novo diagrama de seqüência correspondente ao caso de uso “Gerenciando Ficha Clínica”. Um novo diagrama de transição de estados também é criado.

Com isso podemos novamente acionar o GP, e mais uma vez precisamos recuperar as informações já geradas e que estão em funcionamento na versão anterior do sistema de informação. Recuperamos, como na iteração anterior, todos os métodos definidos e implementados na segunda versão do sistema de informação, com suas interfaces. Importamos a maquete definida para o caso de uso “Gerenciando Ficha Clínica”. E então podemos gerar mais uma versão do sistema de informação. Uma

terceira versão é gerada, mantendo todas as funcionalidades sem perda de informação das versões anteriores, precisando agora, complementar o código gerado, para que os métodos referentes ao novo caso de uso fiquem funcionais.

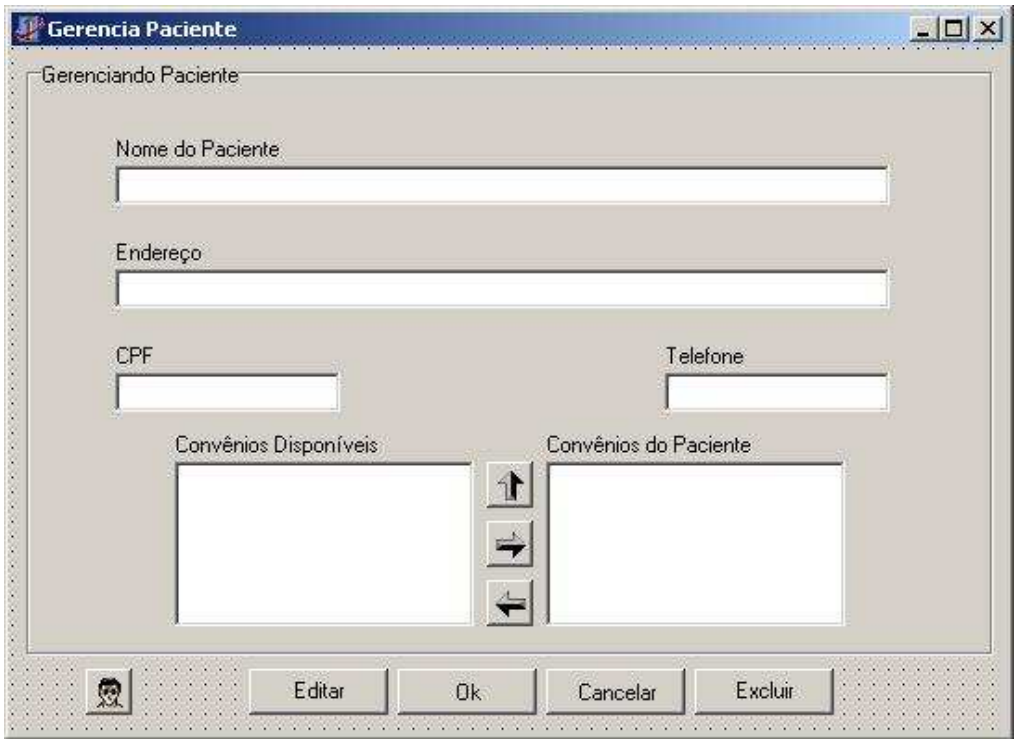

Figura 4-7 Maquete importada para o GP e regerada.

Complementamos então, os métodos do novo caso de uso assim como suas chamadas na classe de controle. Uma nova versão, agora a terceira versão do sistema de informação, é então concluída, e será origem para uma nova iteração do processo.

Percebemos que a partir desse momento o processo se torna repetitivo, não obtendo mais vantagens em continuar com as iterações para o objetivo aqui proposto, pois todos os passos, funcionalidades e processos do GP foram realizados e com total satisfação, mostrando sua eficiência e sua alta produtividade.

Temos a seguir alguns quadros quantitativos que demonstram melhor os resultados alcançados pelo GP.

Número do Passo	Diagramas de Entrada		Units por tipo	
Iteração 1	Casos de Uso	3	Controle	2
	Classes	3	Interface	2
	Sequência	2	Domínio	1
	Estados	1		
Iteração 2	Casos de Uso	6	Controle	4
	Classes	3	Interface	4
	Sequência	4	Domínio	1
	Estados	2		
Iteração 3	Casos de Uso	7	Controle	5
	Classes	3	Interface	5
	Sequência	5	Domínio	1
	Estados	3		

Tabela 4-2 Relação entre os modelos de cada iteração e suas saídas.

Na tabela anterior, vemos a relação entre a quantidade de modelos definidos no FAST CASE e as unidades que são geradas em cada iteração. Optamos por gerar sempre uma unidade de domínio independente da quantidade de classes de domínio definidas. Para cada diagrama de seqüência adicionado, uma classe de controle é adicionada, assim como um novo diagrama de transição de estados é criado, somente os casos de uso do tipo “mantendo”, não vão ter os diagramas de transição de estados, logo uma unidade de controle será criada para cada diagrama de seqüência. Nesse caso cada diagrama de sequência gerou uma nova unidade de interface.

Número do Passo	# de classes do diagrama de classes	
Iteração 1	Classes de Controle	2
	Classes de Interface	2
	Classes de Domínio	2
Iteração 2	Classes de Controle	4
	Classes de Interface	4
	Classes de Domínio	4
Iteração 3	Classes de Controle	5
	Classes de Interface	5
	Classes de Domínio	4

Tabela 4-3 Relação entre as iterações e a quantidade de classes

Nessa outra tabela, podemos observar a evolução do diagrama de classes, com relação a quantidade de classes existentes, durante a evolução do sistema. No final da primeira iteração existem 2 classes de cada tipo. No final da segunda iteração, duas classes de cada tipo foram adicionadas, existindo, portanto 4 classes de cada tipo. Após a terceira iteração, apenas classes de controle e interface são adicionadas, e as classes de domínio se estabilizam. Podemos ver com essa tabela que a tendência no decorrer da evolução do sistema sendo desenvolvido, que as classes de domínio se estabilizam após algumas iterações, e que em cada novo diagrama de seqüência classes de interface e classes de controle serão adicionadas.

Na tabela a seguir, podemos observar a eficiência em relação ao trabalho poupado pelo GP, mostrando a quantidade efetiva de linhas que são geradas em cada iteração e que seriam escritas pelo programador para que o sistema de informação respeitasse a arquitetura proposta.

Número do Passo	Linhas de código geradas pelo GP	Linhas de código introduzidas	Número total de linhas
Iteração 1	609	61	670
Iteração 2	986	53	1039
Iteração 3	1329	41	1370

Tabela 4-4 Relação dos números de linhas geradas

Os objetivos do GP eram dar uma visão de arquitetura de sistemas para o aluno, permitindo a visualização da divisão de responsabilidades entre as classes de um sistema e reduzir o tempo gasto no desenvolvimento de um sistema de informação.

Todos os projetos realizados pelos alunos que utilizaram o GP mantiveram a arquitetura MVC de divisão de responsabilidades entre as classes. Os alunos demonstraram compreender a divisão de responsabilidades entre vários tipos de classes de um sistema através de uma arquitetura padrão. Constatamos que é de suma importância a utilização de ferramentas de geração de código para o desenvolvimento de um sistema de informação. Permitiu também que o aluno abstraisse detalhes da linguagem, e tornou produtivo e proveitoso o aprendizado, permitindo que fosse visto de forma prática conceitos de arquitetura, que poderiam ficar obscuros e incompreensíveis quando mostrados e ensinados somente de forma teórica.

Os projetos desenvolvidos foram concluídos no tempo estimado e disponível para o curso, tornando operacionais as suas funcionalidades, alcançando o objetivo de redução de tempo proposto pelo GP e pelo processo incremental. O objetivo de possibilitar o desenvolvimento de um sistema num pequeno espaço de tempo foi alcançado. A adoção do GP reduz, de maneira significativa, o tempo de desenvolvimento de um sistema orientado a objetos, acarretando assim um aumento significativo na produtividade. Podemos observar pela tabela 4-4, que foram escritas manualmente aproximadamente 150 linhas em uma versão do sistema com 1300 linhas aproximadamente.

Notamos que os alunos têm alguma dificuldade na geração da primeira versão do sistema de informação. Dificuldade esta que é amenizada nas versões seguintes. Constatamos que essas dificuldades são sempre devidas à falta de compreensão das estruturas de mapeamento dos diagramas para o codigo fonte. Mais especificamente estes problemas são:

1- A implementação do mecanismo de “Callback” que possibilita o suporte ao modelo MVC e permite a independência da interface.

2- A especificação do comportamento via o Diagrama de Transição de Estados e o seu mapeamento para o código.

3- A geração de eventos para a classe de controle, utilizando uma interface geradora de eventos apenas.

4- O uso de métodos adequados para escrita e leitura de dados das classes.

5- Enxergar alguns mapeamentos devido a diferença entre os paradigmas OO e relacional.

5 Críticas e Comparações

Existem ferramentas, tanto acadêmicas como comerciais, com o propósito de geração de código. Procuramos listar algumas das suas principais características.

O With Class é uma ferramenta CASE que permite a criação de diagramas orientados a objetos, especificação e geração de código. É baseado na substituição de tags específicas em arquivos textos, utilizados como templates, pré-definidos para cada linguagem de programação específica. (WITH CLASS , 2004)

O ModelMaker (MODELMAKER, 2004) representa uma nova maneira de desenvolver classes e pacotes de componentes para Borland Delphi. A linguagem Object Pascal do Delphi é inteiramente suportada pelo ModelMaker. O ModelMaker possui a capacidade de emgemharia reversa completa. A ferramenta suporta um conjunto de diagramas da UML. A chave mágica do Modelmaker, quanto à velocidade e ao poder da ferramenta é o motor de modelagem ativa que armazena e mantém todos os relacionamentos entre classes e seus membros. Renomeando uma classe ou mudando seu ancestral propagará imediatamente ao código de fonte automaticamente gerado. As tarefas como reescrever os métodos, adicionar eventos, propriedades e os métodos de acesso são reduzidos a selecionar e clicar.

Já o AndroMDA (ANDROMDA, 2004), é um framework de geração de código, baseada em MDA (Model Driven Architeture), que recebe um modelo em UML (Unified Model Lanquage) a partir de uma ferramenta CASE no formato XMI[6] e gera os componentes sob encomenda.

A MVCASE (BARRÉRE, PRADO e BONAFE, 1999), é uma ferramenta CASE

[6] XMI, ou XML Metadata Interchange, é uma especificação que permite facilmente a troca de metadados entre ferramentas de modelagem (baseadas na OMG-UML) e repositórios de metadados (baseado em OMG-MOF) em ambientes heterogêneos.

orientada a objetos, foi desenvolvida com o propósito de investigar a automação do processo de desenvolvimento de sistemas de software, desde a análise e especificação de requisitos, utilizando técnicas de diferentes método, até a implementação automática em linguagem executável e posterior manutenção. Ela é composta por uma ferramenta gráfica, denominada JavaRC, que suporta a especificação de requisitos do sistema usando técnicas de diferente métodos OO, e por um sistema transformacional, denominado Draco, que é responsável pela implementação automática do sistema.

Outro ambiente CASE orientado a objetos, com implementação automática, é composto da ferramenta ToolRC (PRADO *et al.*, 1997) com editor gráfico, que suporta múltiplas visões de requisitos em diferentes técnicas de métodos orientados a objetos, e do sistema transformacional de software Draco, que permite a geração automática de código em C++, a partir de especificações em alto nível de abstração.

O GFMS (FILETO *et al.*, 1996) é um gerador de código fonte, integrante de um ambiente de desenvolvimento de programas aplicativos para o domínio de administração rural. Ele utiliza a abordagem Draco para a construção de software a partir de componentes reusáveis.

O projeto Draco-Puc (BERGMANN, PRADO e LEITE, 1996) é um sistema tranformacional com o objetivo de testar, desenvolver e colocar em prática o paradigma Draco para construção de software. O paradigma Draco é a base da geração código de todos os sistemas citados acima, e a máquina Draco-Puc, é uma implementação parcial do paradigma de desenvolvimento por domínios. Sistemas transformacionais são formados por um conjunto de tecnologias que possibilitam a manipulação simbólica intra e inter-representações. Estas manipulações são executadas através de passos formalmente bem definidos chamados de regras de transformação. Uma regra de transformação é essencialmente formada por dois padrões: o padrão de reconhecimento (LHS) e o padrão de substituição (RHS). Para que uma regra de transformação seja aplicada, o sistema procura o padrão de reconhecimento. Ao ser encontrado, o mesmo é substituído pelo padrão de substituição. As regras podem possuir ainda restrições semânticas, impondo condições que devem ser seguidas antes ou após sua aplicação.

O GP foi desenvolvido com a proposta de interagir com o FAST CASE de uma maneira fácil, e que fosse possível aplicar a arquitetura mencionada na MRDS, o que não existe nas outras ferramentas. As ferramentas existentes e já citadas a cima, não atendem o requerido, além de não aplicar a arquitetura proposta.

6 Conclusões

Os alunos se mostraram satisfeitos com o proposto pelo GP, e deram um retorno positivo, mostrando ser de grande valor a ferramenta desenvolvida e o processo incremental. Este gerador já vem sendo utilizado em turmas da graduação, aonde vêm apresentando como resultados o melhor entendimento dos alunos para a arquitetura de programas e uma redução no tempo gasto em atividades mecânicas de programação, desta forma proporcionando, como esperado, mais tempo livre para a construção dos modelos dos sistemas. O GP está atualmente sendo utilizado por alunos da graduação que estão realizando seus trabalhos de conclusão de curso.

O GP, implementando o mecanismo de *round-trip engineering*, permite com que informações não sejam perdidas no decorrer do desenvolvimento incremental do sistema de informaçao. Esta é uma característica marcante e diferencial para a ferramenta.

Em cada geração sempre é gerada uma estrutura de arquivos Delphi "compiláveis", isto é, sempre que uma nova geração é acionada, o projeto que é gerado é "compilável", gerando um executável que pode ser executado sem nenhum problema.

Os modelos da UML obtiveram um mapeamento padrão através do GP, pois cada modelo e cada diagrama possui uma tradução específica para cada trecho de código gerado, agregando valor semântico aos modelos da UML existentes para o sistema.

O GP gera código para uma arquitetura específica, apresentada pela MRDS, isso é uma característica importante na sua utilização, pois os sistemas que serão gerados pela ferramenta, devem seguir essa arquitetura, não possibilitando a geração de sistemas em arquiteturas diferentes.

De uma forma geral, constatamos em resultados práticos, que a utilização do GP dando apoio a MRDS auxiliou na construção de sistemas de informação, fornecendo

conceitos e vocabulário comuns para o tratamento deste tipo de sistema, alcançando os objetivos propostos.

7 Sugestões de trabalhos futuros

Nesse trabalho a entrada principal de dados do GP são informações provenientes de um conjunto de modelos UML lidas diretamente do repositório do FAST CASE, que segue um formato próprio e não padronizado. Para tornar o GP mais genérico e compatível com outras ferramentas CASE, poderíamos ter como opção a leitura desses modelos UML a partir de arquivos no formato XMI (XML Metadata Interchange).

XMI é uma especificação padrão para descrição de modelos UML definido pela OMG, que é usado como formato de intercâmbio pela maioria dos fabricantes de ferramentas CASE. Com essa facilidade, o GP não só pode ser utilizado por um grande número de ferramentas CASE como também se torna independente do FAST CASE.

Existe ainda a possibilidade de utilizando o modelo já criado, gerar código fonte para outras linguagens de programação. Isto seria possível se o GP gerasse uma linguagem intermediária genérica, que seria traduzível num segundo passo para qualquer outra linguagem alvo.

A consistência do modelo UML que serve de entrada para o processo, pode receber um tratamento de consistência, evitando e alertando o usuário para pontos potenciais de erros de modelagem e consistência.

A geração de casos de teste também é um outro ponto que deixaria o GP mais completo, permitindo que o próprio usuário possa gerá-los e testa-los, tornando a implementação de um sistema uma tarefa mais completa e menos trabalhosa.

8 Apêndices

8.1 Apêndice A

8.1.1 Ordenação Topológica

É realizado no GP uma verificação de consistência básica de verificação de ciclos, para evitar herança cíclica e possível problema na geração. Para isso foi implementado o algoritmo de ordenação topológica nas classes que possuem a relação de herança.

O algoritmo de ordenação topológica, segundo CORMEN, LEISERSON e RIVEST (1991), se baseia no seguinte : suponha que nosso grafo é acíclico. Como é possível tornar a ausência de ciclos evidente? O conceito de ordem topológica responde a essa pergunta. Uma ordem total

$v_1, v_2, \ldots, v_n$

dos vértices de um grafo é **topológica** se todos os arcos apontam no mesmo sentido. Mais especificamente, a ordem é topológica se

não existe arco (v_i, v_j) com $i <= j$,

ou seja, se todos os arcos apontam "pra trás". A definição de CORMEN, LEISERSON e RIVEST (1991) diz que todos os arcos devem apontar "pra frente", mas no fundo estamos falando da mesma coisa.

É evidente que um grafo dotado de um ciclo não possui ordenação topológica (e portanto, a existência de ordenação topológica prova que o grafo é acíclico).

8.2 Apêndice B

O Delphi possui uma hierarquia de classes. Qualquer classe no Delphi é uma subclasse da classe TObject, deste modo toda a hierarquia possui somente uma única raiz.

A VCL (visual component library), que é a biblioteca de componentes visuais do Delphi, define várias subclasses de TObject. Muitas dessas classes são realmente subclasses de outras subclasses, formando uma hierarquia muito complexa.

Cada classe da VCL possui suas propriedades e seus métodos, mas algumas propriedades, eventos e métodos das classes VCL estão definidos em um nível mais alto da hierarquia de classe, e, portanto, estão disponíveis para qualquer componente. A seguir estão as propriedades, eventos e métodos comuns (SCHMITZ e SILVEIRA, 2000).

Principais Componentes (Propriedades, Eventos e Métodos)

Propriedades Comuns

Nome	Descrição
Align	Determina como o componente será alinhado no seu container [alNone, alTop, alBottom, alLeft, alRight, alClient];
Caption	Legenda do componente;
Cursor	Desenho que representa o cursor da mouse [crDefault, crNone, crArrow, crCross, crIBeam, crSize, crSizeNESW, crSizeNS, crSizeNWSE, crSizeWE, crUpArrow, crHourGlass, crDrag, crNoDrop, crHSplit,

	crVSplit, crMultiDrag, crSQLWait, crNo, crAppStart, crHelp, crHandPoint];
Name	Nome da instância do componente;
Left	Distância em Pixel da borda esquerda do componente até a borda esquerda do FORM;
Top	Distância em pixel da borda superior do componente até a borda superior do FORM;
Height	Altura em pixel do componente;
Width	Largura em pixel do componente;
Componentcount	O número de componentes possuídos por um componente container;
Components	Uma matriz de componentes possuídos por um componente container;
Color	Indica a cor de fundo do componente;
Font	Fonte utilizada no componente;
ctl3d	Define a aparência 3D ou 2D de um componente;
Enabled	Define se o componente esta ou não ativo;
Visible	Define se o componente esta ou não visível;
Hint	String utilizada na exibição de dicas instantâneas;
Showhint	Define se o hint será mostrado automaticamente;
Popupmenu	Menu que será acionado com o botão direito do mouse;

Taborder	A ordem de tabulação do componente;
Tabstop	Indica se o componente será ponto de parada para a tecla TAB;
Helpcontext	Número utilizado para chamar o help on-line sensível ao contexto;

Eventos Comuns

Nome	***Descrição***
OnChage	O conteúdo do componente é alterado;
OnClick	O componente é acionado (Mouse ou Enter);
OnDblClick	Duplo-clique sobre o componente;
OnEnter	O componente recebe o foco;
OnExit	O componente perde o foco;
OnKeyDown	Tecla(s) são pressionada(s) inclusive teclas de controle Parametros : *Key* = Código da tecla *Shift* = conjunto que indica a(s) tecla(s) de controle pressionadas (*ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble*);
OnKeyPress	Uma tecla é pressionada, onde *Key* contém o caracter pressionado;
OnKeyUp	Uma tecla é solta;

Métodos Comuns

Nome	***Descrição***

Create	Cria uma nova instância;
Destroy	utilizado para destruir uma instância;
Show	Torna o componente visível;
Hide	Torna o componente invisível;
SetFocus	Coloca o foco no componente;
Focused	Determina se o componente tem o foco;
BringToFront	Coloca o componente na frente dos outros;
SendToBack	Coloca o componente atrás dos outros;
ScaleBy	Gradua o componente em determina escala. Ex: Button1.ScaleBy(80,100) altera o tamanho do botão para 80% do tamanho original;
SetBounds	Muda a posição e o tamanho do componente (*ALeft, ATop, AWidth, AHeigh*);

TForm

Elemento básico no desenvolvimento ***Delphi*** formando o alicerce sobre, o qual um aplicativo é construído. Todo o componente que você venha a colocar em um formulário, ou qualquer propriedade que você estabeleça, é armazenado em um arquivo que descreve o formulário (um arquivo do tipo .DFM) e também exerce algum efeito sobre o código-fonte associado ao formulário (o arquivo .PAS).

Propriedades	***Descrição***
Active	Indica quando o formulário esta ativo;
ActiveControl	Determina o controle que receberá o foco quando o

	formulário é ativado;
AutoScroll	Adiciona barras de rolagens automaticamente quando um formulário é redimensionado de forma a cobrir componentes;
HorzScrollBar	Adiciona Barra de rolagem Horizontais quando necessário;
VertScrollBar	Adiciona Barra de rolagem Verticais quando necessário;
BorderIcons	Define quais ícones de controle serão visíveis [*biSystemMenu, biMinimize, biMaximize, biHelp*];
BorderStyle	Estilo da borda da janela [*bsDialog, bsSingle, bsNone, bsSizeable, bsToolWindow, bsSizeToolWin*];
FormStyle	Tipo da janela [*fsNormalfs, MDIChild, fsMDIForm, fsStayOnTop*];
Icon	Ícone da janela;
Menu	Indica qual o componente menu do formulário será apresentado;
Position	Permite controlar a posição e tamanho dos formulários na execução [*poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter*];
WindowState	Estado da janela (normal, minimizado ou maximizado);

Eventos	***Descrição***

OnCreate	O formulário é criado;
OnShow	Antes de mostrar a janela;
OnCloseQuery	É chamada para validar se a janela pode ser fechada;
OnClose	Ocorre quando a janela é fechada;
OnActivate	Ocorre quando a janela torna-se ativa;
OnDeactivate	Ocorre quando a janela perde o foco;
OnResize	Ocorre quando a janela muda de tamanho;

Métodos	***Descrição***
Show	Mostra uma janela não-modal;
ShowModal	Ativa uma janela modal;
Close	Fecha a janela;
Refresh	Redesenha a janela;

TButton

Botão que permite a sinalização de um evento através do seu pressionamento.

Propriedades	***Descrição***
Cancel	Dispara o evento *OnClick* do botão quando a tecla ESC é pressionada;

Default	Dispara o evento *OnClick* do botão quando a tecla ENTER é pressionada;
ModalResult	Associa o botão a opção de fechamento de um *Form* modal;

TBitBtn

Similar ao *Button*, mas podendo mostrar uma imagem.

Propriedades	***Descrição***
Kind	Seleciona um *BitMap* padrão para o botão;
Style	Indica a aparência do botão (*win3.11, win95, win98, winxx*);

TStrings

Muitos componentes possuem propriedades do tipo *TStrings*, essa classe permite armazenar e manipular uma lista de Strings. Toda propriedade do tipo *TStrings* permite acesso indexado aos itens da listas (Propriedade[indíce]).

É uma classe abstrata para representar todas as formas de lista de *strings*, independentemente de sua implementações de armazenamento. Os objetos instanciados a partir de *TStrings* são utilizados somente como propriedades de componentes que necessitam armazenar *strings*.

Propriedades	***Descrição***
Count	Número de linhas;

Métodos	*Descrição*
Add	Adiciona uma nova linha no final da lista;
Insert	Insere uma nova linha numa posição especificada;
Delete	Apaga uma linha;
Clear	Apaga toda a lista;
IndexOf	Retorna o índice do item e –1 caso não encontre;
LoadFromFile	Carrega de um arquivo texto;
SaveToFile	Salva para um arquivo texto;

TMainMenu e TPopupMenu

Usado para criação de barras de menus, onde o MainMenu representa o Menu suspenso presente nos aplicativos e o PopupMenu representa o menu instantâneo (botão direito do mouse) associado aos componentes através da propriedade PopupMenu.

Propriedades	*Descrição*
Items	Itens do menu, utiliza o *MenuEditor*;

Eventos	*Descrição*
OnPopUp	Ocorre quando o menu é ativado;

O ***MenuEditor*** é utilizado para definição dos itens e sub-itens do menu. As suas operações principais são: inserir, deletar.

TMenuItem

Propriedades	***Descrição***
Cpation	Texto associado ao item;
Checked	Indica se o item está ou não selecionado;
Visible	Indica se o item está ou não visível;
Enabled	Indica se o item está ou não ativado;
ShortCut	Tecla de atalho;
Break	Indica quebra de coluna;
GroupIndex	Indica que o item pertence a um determinado grupo;
RadioItem	Indica que o item de menu funcionará como um *Radio*, ou seja, dentro do mesmo grupo apenas um estará selecionado;

Eventos	***Descrição***
OnClick	Ocorre quando o item de menu é selecionado;

TTable

Propriedades	*Descrição*
Active	Define se a tabela esta aberta ou fechada;
DatabaseName	Nome lógico do banco de dados ou nome de um diretório;
TableName	Nome da tabela;
ReadOnly	Define se pode ser feita atualização na tabela;
Exclusive	Define se a tabela pode ser compartilhada;
BOF	Informa se está no início da tabela;
EOF	Informa se está no fim da tabela;
IndexName	Nome do índice em uso;
IndexFieldNames	Nome dos campos de ordenação;
MasterSource	Nome do DataSource mestre em um relacionamento;
MasterField	Campos do relacionamento;
State	Estado da tabela (Inativa, Edição, Inserção, Pesquisa, Cálculo de Campo);
RecordCount	Número de registros;
Fields[n]	Faz referência ao n-ésimo campo da Tabela;

Métodos	*Descrição*
Open	Abre a tabela;

Close	Fecha a tabela;
FieldByName	Faz referência a campo através do nome;
First	Vai para o primeiro registro;
Next	Vai para o próximo registro;
Last	Vai para o último registro;
Prior	Vai para o registro anterior;
Insert	Coloca a tabela no modo de inserção;
Delete	Remove o registro corrente;
Edit	Coloca a tabela em modo de edição;
Post	Grava o registro corrente;
Cancel	Cancela a operação atual (inserção ou alteração);
FindKey	Procura um registro pela chave;
FindNearest	Procura um registro pela chave posicionando no mais próximo;
SetKey	Entra em modo de procura;
GotoKey	Procura um registro pela chave;
GotoNearest	Procura um registro pela chave posicionando no mais próximo;
EmptyTable	Apaga todos os registro da tabela, para isso a tabela não pode esta sendo compartilhada;

Evento	*Descrição*
BeforeOpen	Ocorre no início da execução do método *Open*;
AfterOpen	Ocorre após a execução do método *Open*;
BeforeClose	Ocorre antes do fechamento da Tabela;
AfterClose	Ocorre após o fechamento da Tabela;
BeforeInsert	Ocorre no início da execução do método I*nsert*;
AfterInsert	Ocorre após a execução do método *Insert*;
BeforeEdit	Ocorre no início da execução do método *Edit*;
AfterEdit	Ocorre após a execução do método *Edit*;
BeforeDelete	Ocorre no início da execução do método *Delete*;
AfterDelete	Ocorre após a execução do Método *Delete*;
BeforeCancel	Ocorre no início da execução do método *Cancel*;
AfterCancel	Ocorre após a execução do método *Cancel*;
BeforePost	Ocorre no início da execução do método *Post*;
AfterPost	Ocorre após a execução do método *Post*;
OnNewRecord	Ocorre quando um novo registro é adicionado a tabela;
OnCalcField	Ocorre quando é necessário saber o valor de um campo calculado;

TDataSource

Propriedade	*Descrição*
DataSet	*Query* ou *Table* ao qual faz referência;

Evento	*Descrição*
OnDataChange	Ocorre quando o DataSet ou qualquer dos controles é alterado;
OnStateChange	Ocorre quando o estado do DataSet é alterado;
OnUpdateData	Ocorre antes de uma atualização;

DataModule

Permite agrupar os componentes *Table*, *Query* e *DataSource* evitando a repetição destes em diferentes formulários de uma mesma aplicação. Depois de inserir um *DataModule* na Aplicação [*File/NewDataModule*], insira, neste, os componentes *Table*, *Query* e *DataSource*, em seguida em qualquer formulário que possua *DataControl's* faça referência os Componentes *Table*, *Query* e *DataSource* indicando antes o nome do *DataModule*.

TDBGrid

Propriedades	*Descrição*
DataSource	*DataSource* ao qual o controle está ligado;

TDBNavigator

Propriedades	*Descrição*
DataSource	*DataSource* ao qual o controle está ligado;
VisibleButtons	Define os botões que serão visíveis;
ConfirmDelete	Define se será solicitado uma confirmação antes da exclusão;

Field

Propriedades	*Descrição*
FieldName	Campo ao qual o controle está ligado;
DataType	Tipo do campo armazenado (*ftBoolean*, *ftBCD*, *ftBlob*, *ftBytes*, *ftCurrency*, *ftDate*, *ftDateTime*, *ftFloat*, *ftGraphic*, *ftInteger*, *ftMemo*, *ftSmallint*, *ftString*, *ftTime*, *ftUnknown*, *ftVarBytes*, ou *ftWord*);
Value	Valor do campo;
IsNull	Indica se o valor do campo é nulo;
EditMask ou EditFormat	Máscara de edição que será utilizada na alteração;
DisplayLabel	Título a ser exibido para o campo;
Size	Tamanho do campo;
Calculated	Indica se o campo é calculado em tempo de

	execução;
Required	Define se o campo é obrigatório;
DisplayFormat	Máscara que será utilizada na exibição;
MaxValue	Valor máximo para o campo;
MinValue	Valor mínimo para o campo;
AsString, AsInteger, AsFloat, AsBoolean, AsDateTime	Acessa o campo no respectivo formato;

Eventos	***Descrição***
OnValidate	Ocorre sempre que o campo é armazenado na tabela

Método	***Descrição***
Clear	Limpa o campo

8.3 Apêndice C

8.3.1 Tipos Procedurais

Ponteiros para métodos é um subconjunto de um tipo que o *Dephi* chama de tipo procedural. Esse tipo é utilizado na arquitetura padrão para a tratar os eventos da interface, por permitir menor acoplamento entre as classes de interface e controle, pois a interface gera eventos que são tratados pelo controle, sem a interface "conhecer" a classe de controle.

Em Delphi, podemos invocar métodos de três modos (PINTO e BASTOS, 2000):

- referências de classes,
- referências de objetos, e
- ponteiros de métodos.

Uma referência de classe pode ser usada para invocar métodos que se aplicam a uma classe inteira (métodos de classe). Referências de objeto e ponteiros de método podem ser usados para invocar qualquer método da classe, ou métodos que pertencem a um exemplo de objeto específico (métodos de objeto).(PINTO e BASTOS, 2000)

Invocar um método por referência de objeto ou referência de classe é mais comum e mais fácil entender, particularmente para o novato em Delphi. Quando um método é invocado por referência, o nome do método - como definido na classe do objeto - é declarado explicitamente junto com os parâmetros apropriados (se existir). Por exemplo, o código a seguir invoca um método por uma referência de objeto (PINTO e BASTOS, 2000):

```
meuObject.UmMetodo (Parâmetro);
```

O exemplo a seguir invoca um método por referência de classe (a classe é

TEdit). Neste caso, o constructor é invocado ao nível de classe para criar uma nova instância da classe:

```
myEdit := TEdit.Create(nil);
```

Figura 8-1 Invocação de método por referência de classe.

Um ponteiro de método, como indica seu nome, pode conter o endereço de um objeto ou método de classe (PINTO e BASTOS, 2000). Como declarado na Ajuda on-line do Delphi: “Delphi lhe permite declarar tipos procedurais que são métodos de objeto, permitindo a chamada de métodos particulares de objetos particulares em tempo de execução.”

Tipos procedurais lhe permitem tratar procedimentos e funções como valores que podem ser nomeados a variáveis ou podem ser passados a outros procedimentos e funções. Por exemplo, suponha você define uma função chamada Calc que tem dois parâmetros inteiros, e retorna um inteiro, você pode associar a função Calc a variável F (PINTO e BASTOS, 2000):

```
function Calc(X,Y: Integer): Integer;
...
var F: function(X,Y: Integer): Integer;
...
F := Calc;
```

Figura 8-2 Tipos procedurais.

Se você pega qualquer procedimento ou função e remove o identificador que está depois da palavra procedure ou function, o que está na esquerda é o nome do tipo procedural. Você pode usar tal tipo para nomear diretamente as declarações das variáveis (como no exemplo acima) ou declarar tipos novos (PINTO e BASTOS, 2000):

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var  F: TIntegerFunction;      {F é uma função sem parâmetro que retorna um interiro}
  Proc: TProcedure;          {Proc é um procedimento sem parâmetro}
  SP: TStrProc;              {SP é um procedimento que tem como parâmetro uma string}
  M: TMathFunc;              {M é uma função que tem um Double como parâmetro
                             e retorna um Double}
procedure FuncProc(P: TIntegerFunction);  {FuncProc é um procedimento que tem como
parâmetro uma função, que retorna um inteiro, sem parâmetros}
```

Figura 8-3 Declarações de vários tipos procedurais

As variáveis acima são todas ponteiros de procedimentos – que são ponteiros para o endereço de um procedimento ou função. Se você quer a referência de um método da instância de uma classe (tipo), você precisa acrescentar as palavras of object após o nome do tipo procedural.

Estes tipos representam ponteiros para métodos. Um ponteiro de método é na verdade um par de ponteiros; o primeiro guarda o endereço de um método, e o segundo guarda uma referência para o objeto que o método pertence. Podemos realizar a seguinte atribuição:

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ...
  end;
var
  MainForm: TMainForm;
  OnClick: TnotifyEvent
...
OnClick := MainForm.ButtonClick;
```

Figura 8-4 Ponteiros para métodos

Uma variável do tipo procedural (por exemplo, um ponteiro de método) pode ser associado a qualquer objeto ou método de classe que tenha a mesma assinatura. Uma vez a associação senda feita, o ponteiro de método pode ser usado para invocar o método (PINTO e BASTOS, 2000).

Note que a assinatura de uma função ou procedimento está definida por sua lista de argumentos (os tipos dos argumentos) junto com, no caso de uma função, o tipo de valor de retorno (PINTO e BASTOS, 2000).

Tipos de ponteiro de procedimento sempre são incompatíveis com tipos de ponteiro de método. O valor **nil** pode ser associado a qualquer tipo procedural (PINTO e BASTOS, 2000).

Procedimentos aninhados e funções (rotinas declaradas dentro de outras rotinas) não podem ser usadas como valores procedurais, nem funções e procedimentos predefinidos. Se você quiser usar a rotina predefinida Length como um valor procedural, escreva um wrapper para isto:

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

Figura 8-5 Wrapper para rotinas predefinidas

9 Referências Bibliográficas

AMBLER, Scott W., “Mapping objects to relational databases – What you need to know and why”, Julho 2000.

_____, “Mapping Objects To Relational Databases”, Outubro 2000.

ANDROMDA; http://www.andromda.org/, acessado em março de 2004.

BARRÉRE, T. S., PRADO, A.F., BONAFE, V.C., CASE Orientada a Objetos com Múltiplas Visões e Implementação Automática de Sistemas – MVCASE, Anais do XIII Simpósio Brasileiro de Engenharia de Software, Ed. SBC, 1999, pp.113-128.

BASS, Len, CLEMENTS, Paul, KAZMAN, Rick; “Software Architecture in Pratice”, Addison Wesley, 1998

BERGMANN, U., PRADO, A.F., LEITE, J.C.S.P, Desenvolvimento de Sistemas Orientado a Objetos Utilizando o Sistema Transformacional Draco-Puc, Anais do X Simpósio Brasileiro de Engenharia de Software, Ed. SBC, 1996, pp.173-188.

BOOCH, G., JACOBSON, I., RUMBAUGH, J. The Unified Modeling Language Reference Manual, Addison Wesley Object Technology Series 1999.

BROERING, E.; “MDA - Model Driven Architeture”; http://www.portaljava.com.br/home/modules.php?name=Content&pa=showpage&pid=14, acessado em março de 2004.

BURBECK, Steve, Ph.D. “Applications Programming in Smalltalk-80ª: How to use Model-View-Controller (MVC)”.

CLEAVELAND. C.; “Building Application Generators”; IEEE Software 5(4); 1988; pp 25-33

CLEMENTS, Paul C., “Coming Attractions in Software Architecture” , 1996, Software

Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

CORMEN, Th.H., LEISERSON Ch.E., RIVEST, R.L., “Introduction to Algorithms”, MIT Press & McGraw-Hill, 1991.

DASS, Karan; “Model-View-Controller (MVC) Architecture”, http://www.indiawebdevelopers.com/technology/java/mvcarchitecture.asp, acessado em março de 2003.

DAVIS, Malcolm; “Struts, an open-source MVC implementation - Manage complexity in large Web sites with this servlets and JSP framework”, 2001, http://www-106.ibm.com/developerworks/java/library/j-struts/, acessado em março de 2004.

ESPERANÇO, Cláudio P., FILHO, José Luiz R. D.,SILVEIRA, Denis e SCHMITZ, Eber, “Engenharia Reversa em Sistemas de Informação Utilizando XMI”, II Conferência da Associação Portuguesa de Sistemas de Informação, Portugal, Novembro 2001.

FILETO, R., MEIRA, C.A.A., COSTA, C.R., MASSHURÁ, S.M.F.S., A Construção de um Gerador de Programas Aplicativos Segundo Conceitos de Análise de Domínios, Anais do X Simpósio Brasileiro de Engenharia de Software, Ed. SBC, 1996, pp.119-135.

FRANCA, Luiz Paulo Alves, “Um processo paa construção de geradores de artefatos”, Tese de Doutorado, Puc-RJ, 2000.

GARLAN et al., “Architectural Styles, Design Patterns, and Objects”, IEEE Software, pp.43-52, Jan/Fev 1997.

GARLAN, D. e SHAW, M.; “An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering”, volume I. World Scientific Publishing Company, River Edge, NJ, 1993.

GHEZZI, C., JAZAYERI, M., MANDRIOLI, D., Fundamentals of Software, Engineering, Prentice Hall, Inc., 1991

GONÇALVES, Fátima C. V. e FAÇANHA, Raquel L., “Geração Automática de Código para Classes de Domínio”, Universidade Federal do Rio de Janeiro, Projeto Final do curso de Bacharel em Informática, 2000/2

HOFMEISTER, Christine, NORD, Robert, SONI, Dilip; “Applied Software Architecture, Addison Wesley”, 2000.

HOHENSTEIN, U.; “An Approach for Generating Object-Oriented Interfaces for Relational Databases”; In Proceedings of the Second International Symposium on Constructing Software Engineering Tools; 2000; pp 101-111.

JACOBSON et al., “Object-Oriented Software Engineering – A Use Case Driven Approach”, Addison Wesley, 1992.

JENKINS, M.; “Surveying The Software Generator Market”; Datamation; Sept. 1985; pp 105-120.

MASIERO, P.C.; MEIRA, C. A.; “Development and Instantiation of a Generic Application Generator”; J. System Software; 23; 1993; pp 27-37

MELLOR, Stephen J., BALCER, Marc J.; “Executable UML - A Foundation for Model-Driven Architecture”, Addison-Wesley, 2002.

MILICEV, D.; “Extended Object Diagrams for Transformational Specifications in Modeling Enviroments”; In Proceedings of the Second International Symposium on Constructing Software Engineering Tools; 2000; pp 121-131.

MODELMAKER; http://www.modelmakertools.com/, acessado em março de 2004.

NEIGHBORS, J. M., BIGGERSTAFF, T., PERLIS, A.; “Draco: A Method for Engineering Reusable Software Systems”; Software Reusability; Addison-Wesley/ACM Press; 1989; pp 295-319

OMG, “MDA Guide version 1.0.1”, Document Number: omg/2003-06-01, 12 de junho de 2003.

"OMG Unified Modeling Language Specification", versão 1.5, OMG, Fevereiro 2004. http://cgi.omg.org/cgi-bin/doc?formal/01-09-67.

PFLEEGER, S.L.; Software Engineering Theory and Practice; Prentice-Hall; NL; 1998.

PINTO, Márcio A. O., SILVEIRA, Denis e SCHMITZ, Eber, "Um Gerador de Programas – FAST CASE", XIV Simpósio Brasileiro em Engenharia de Software, Outubro 2000 (João Pessoa/PB)

PINTO, Márcio A. O., BASTOS, Renato N., "O Gerador Automático de Programas do Fast Case", Projeto Final de Curso, DCC/UFRJ, 2000

PRADO, A.F., COUTO, A.A.C.R., LIMA, M.A.V., SILVA, T.E., ToolRC : Ferramenta CASE Orientada a Objetos, Anais do XI Simpósio Brasileiro de Engenharia de Software, Ed. SBC, 1997, pp.503-506.

PRESSMAN, R. S., Software Engineering – A Practitioner's Approach, 5a Edition, McGraw-Hill series in computer science, 2001;

PROJECT, "Project Technology", http://www.projtech.com/info/mda.html, acessado em março de 2004.

SCHMITZ, Eber e SILVEIRA, Denis, "Desenvolvimento de Software Orientado a Objetos", Julho de 2000, Ed. BRASPORT.

_____, "FAST CASE: Uma Ferramenta para o Desenvolvimento Visual de Sistemas Orientados a Objetos", XIII SBES, Caderno de Ferramentas, pp. 29-32, 1999

SELIC, B., "Complete High-Performance Code Generation From Uml Models", Proceedings, Embedded Systems Conference Spring, 2000, Boston MA (CMP Media LLC). (http://www.esconline.com/db_area/00spring/328.pdf)

SHAW, Mary e GARLAN, David; "Software Architecture. Perspectives on an Emerging Discipline", Prentice Hall, 1996.

SILVEIRA, D., “FAST CASE: Uma Ferramenta para o Desenvolvimento Visual de Sistemas Orientados a Objetos”, Tese de Mestrado, IM/NCE/UFRJ, 1999

SILVEIRA, Denis e SCHMITZ, Eber; “MRDS: Uma Metodologia de Desenvolvimento de Software em Empresas de Pequeno e Médio Porte”, II Conferência da Associação Portuguesa de Sistemas de Informação, Portugal, Novembro 2001;

SINGH, Inderjeet, STEARNS, Beth, JOHNSON, Mark, and the Enterprise Team, “Designing Enterprise Applications with the J2EE™ Platform”, Second Edition, 2002

STAA, A.; Manual de Referência: Talisman: Ambiente de Engenharia de Software Assistido por Computador; Rio de Janeiro; 1993.

STAA, A. V., “Programação modular - Desenvolvendo programas complexos de forma organizada e segura”, Editora Campus 2000.

VAROTO, Ane C.; “Visões em Arquitetura de Software”, Dissertação de Mestrado, Instituto de Matemática e Estatística da USP, Março 2002.

WITH CLASS; http://www.microgold.com/ acessado em março de 2004.

ZIMBRÃO, G.; “Mapeando um modelo de classes para um banco de dados relacional”; SQL Magazine 2003.