UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uma Abordagem para Construção e Reutilização de Componentes de Software com implementação em Delphi

João Luís Cardoso de Moraes

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

São Carlos
Julho/04

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

M827ac

Moraes, João Luis Cardoso de.
Uma abordagem para construção e reutilização de componentes de software com implementação em Delphi / João Luis Cardoso de Moraes. -- São Carlos : UFSCar, 2005.
128p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Análise e projeto de sistemas. 2. Sistema de transformação. 3. Componentes de software. 4. Padrões de projeto. I. Título.

CDD: 004.2 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Uma abordagem para Construção e Reutilização de Componentes de Software com Implementação em Delphi"

JOÃO LUIS CARDOSO DE MORAES

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Antonio Francisco do Prado
(Orientador - DC/UFSCar)

Prof. Dr. Sérgio Donizetti Zorzo
(DC/UFSCar)

Prof. Dr. Ulf Bergmann
(IME/RJ)

São Carlos
Agosto/2004

Sumário

Lista de Figuras e Tabelas

Lista de Figuras

Lista de Tabelas

Agradecimento

Agradeço imensamente à minha esposa ELAINE e aos meus filhos KAROLINE e JOÃO LUÍS pela maturidade demonstrada durante esta longa caminhada.

Agradeço a DEUS, o Arquiteto do Universo, pelo apoio recebido durante esta longa caminhada. Sempre esteve do nosso lado, olhando nossos passos.

Resumo

Este projeto desenvolve uma abordagem para construção e reutilização de componentes de software com implementação em *Delphi*.

Cada vez mais se emprega recursos computacionais na solução de problemas, com o objetivo de automatizá-los, para diminuir custos e aumentar a eficiência, desempenho, e segurança. Para atender esta demanda de sistemas de softwares mais confiáveis, mais eficientes e com menores custos, novos métodos têm sido pesquisados, destacando-se, atualmente, o Desenvolvimento Baseado em Componentes (DBC).

São utilizadas técnicas de DBC, Padrões de Projeto, Componentes de Software, a Linguagem de Modelagem *MDL*, e a Linguagem *ObjectPascal (Delphi),* como controle para a abordagem proposta. Os mecanismos utilizados são: ferramenta MVCASE, ST Draco-PUC, engenheiro de software e o ambiente *Delphi*. Esses mecanismos integrados orientam o engenheiro de software tanto na construção de componentes quanto da sua reutilização para criação das aplicações, e a ferramenta MVCASE automatiza parte das tarefas de construção e reutilização de componentes.

A abordagem está dividida em duas etapas, *Construir Componentes* e *Reutilizar Componentes*. Numa primeira etapa, o engenheiro de software parte dos requisitos de um domínio do problema e produz o projeto dos componentes de um domínio e os componentes instalados no ambiente *Delphi*. Uma vez instalados, os componentes serão armazenados num repositório para posterior reutilização.

Numa etapa subseqüente o engenheiro de software consulta os componentes de um domínio do problema, que são importados do repositório para a MVCASE, ficando disponíveis no *browser* da ferramenta. Após a identificação dos componentes necessários, constroem-se as aplicações que os reutilizam.

Os componentes podem ser implementados em Linguagem Orientada a Componentes como C++, Java, e *ObjectPascal* (*Delphi*). Neste projeto adotou-se a Linguagem *ObjectPascal* para implementação dos componentes.

Abstract

This project searches a boarding for construction and reuse of software components with implementation in *Delphi*.

More and more computational resources are used in the solution of problems, aiming to automatize them, to reduce costs and increase the efficiency, performance, and security. To take care of this demand of more trustworthy softwares systems, more efficient and with lesser costs, new methods have been searched, being distinguished, currently, the Component-Based Development (CBD).

The research studies different methods, available techniques and tools in literature as: CBD, Design Patterns, Software Components, Model Language (MDL), and *ObjectPascal* Language (*Delphi*), as control for the boarding. The mechanism utilized are: MVCASE tool, Draco-PUC Transformation System (TS), Software Engineer, and *Delphi* (RAD). This integrated mechanisms orient the Software Engineer on the Construct and on the Components Reuse for aplication development, and MVCASE tool automatize the tasks for components construct and reuse.

The boarding is divided in two stages: Construct Components and Reuse Components. In a first stage, the software engineer initial from the problem domain knowledge and produces a models components of the domain, implemented in a language oriented-components, in the case the *Delphi*. As soon as they are instaled, the components are keep in the repository for reuse.

In a subsequent stage the software engineer will find the components about the problem domain, that are imported from MVCASE tool repository, where they are show in tool browser. After the components are identified the applications are developed that reuse the framework components constructed.

The components may be implemented in language components oriented like, C++, Java, and *ObejctPascal* (*Delphi*). In this project we are using the *ObjectPascal* language for the components implementation.

Capítulo 1

Introdução

1.1 Considerações Iniciais

No desenvolvimento de software, a reutilização caracteriza-se pela utilização de produtos de software, em uma situação diferente daquela para qual estes produtos foram originalmente construídos [Werner, 2000]. O Desenvolvimento Baseado em Componentes (DBC) se preocupa com a criação de componentes que possam ser reutilizados em outras aplicações. Como solução para este problema, pesquisas [D'Souza, 1998], [Werner, 2000], [Pressman, 2001] apontam, como passo fundamental, a sistematização do processo de análise e criação de componentes para um determinado domínio de aplicações.

Para que a reutilização possa ser efetiva, deve-se considerá-la em todas as fases do processo de desenvolvimento do software. Portanto, o Desenvolvimento Baseado em Componentes (DBC) deve oferecer métodos, técnicas e ferramentas que suportem desde a identificação e especificação dos componentes, referente ao domínio do problema, até o seu projeto e implementação em uma linguagem executável. Além disso, o DBC deve empregar inter-relações entre componentes já existentes, que tenham sido previamente testados, visando reduzir a complexidade e o custo de desenvolvimento do software [Werner, 2000].

Apesar das recentes e constantes pesquisas na área de DBC, ainda há carência de métodos, técnicas e ferramentas que suportam tanto o desenvolvimento quanto a reutilização de componentes em aplicações de um determinado domínio. Outro problema importante relaciona-se com a dificuldade de integração das diferentes técnicas e ferramentas que apóiam o DBC.

A reutilização é um princípio essencial na área de Engenharia de Software para garantir a redução de esforços e custos no desenvolvimento de software e a redundância de código. Na tecnologia orientada a objetos, a reutilização de software pode ser assegurada com a adoção de *patterns*, *frameworks* de domínios específicos e componentes de software já existentes e testados.

Pesquisas têm explorado diferentes tecnologias, incluindo o uso de ferramentas CASE (*Computer-Aided Software Engineering*), componentes de software, engenharia de domínio (ED), *frameworks*, padrões, Sistemas de Transformação de Software e linguagens de programação, orientadas a componentes, para obter software de melhor qualidade e com menor custo.

1.2 Objetivos

A abordagem está dividida em duas etapas, *Construir Componentes* e *Reutilizar Componentes*. Numa primeira etapa, o engenheiro de software parte do estudo do conhecimento de um domínio do problema e produz o projeto dos componentes de um domínio específico e os componentes implementados em uma linguagem orientada a componentes. Uma vez instalados, os componentes serão armazenados num repositório para posterior reutilização.

Numa etapa subseqüente o engenheiro de software consulta os componentes de um domínio do problema, que são importados do repositório para ferramenta MVCASE, ficando disponíveis no *browser*. Após a identificação dos componentes necessários, constroem-se as aplicações que os reutilizam.

Para a construção de componentes a abordagem compreende 3 passos: *Modelar Componentes, Gerar Código dos Componentes e Instalar Componentes*. O passo Modelar Componentes é subdividido em 3 passos: *Definir Domínio do Problema, Especificar Componentes e Projetar Componentes*.

No passo *Definir Domínio do Problema* é dado ênfase no entendimento do domínio do problema, especificando-se "o quê" os componentes devem atender para solucionar o problema. No passo *Especificar Componentes* são descritos os comportamentos externos do componente de uma forma não ambígua. Em seguida, no passo *Projetar Componentes,* o Engenheiro de Software realiza o projeto interno dos componentes, e especifica outros requisitos não funcionais, destacando-se a persistência dos componentes, e são gerados os métodos dos componentes com implementação em *ObjectPascal*, sendo utilizada a ferramenta MVCASE como principal mecanismo.

Com base no Projeto Interno dos Componentes, no passo *Gerar Código dos Componentes* é gerado o código dos componentes, na linguagem *ObjectPascal,* a partir das Especificações *MDL* obtidas no passo anterior, usando o Sistema Transformacional Draco-PUC como mecanismo.

Após a geração do código o engenheiro de software irá, no último passo da construção dos componentes, instalar os componentes no ambiente *Delphi*, sendo necessário compilar os pacotes gerados.

Uma vez construídos os componentes, estes podem ser reutilizados em nível de projeto e de implementação.

Na abordagem, para construir as aplicações reutilizando os componentes, deve-se: *Modelar a Aplicação*, a partir da documentação dos componentes, disponível na ferramenta MVCASE; *Gerar Código da Aplicação,* reutilizando os componentes; *Executar a Aplicação* no ambiente *Delphi*, e *Refinar o código da aplicação* no ambiente *Delphi*, para atender aos requisitos não funcionais, específicos para tratamento de crítica de dados, acesso a banco de dados, e outros aspectos.

1.3 Motivação

Motivado pelas idéias de construção e reutilização de componentes de software, através do uso de várias técnicas que visam reaproveitar ao máximo o trabalho de análise, projeto e implementação já concluído, e pelo desenvolvimento baseado em componentes, este projeto desenvolve uma abordagem para a construção e reutilização de componentes implementados em uma linguagem orientada a componentes, que procura orientar o engenheiro de software tanto no desenvolvimento dos componentes de um domínio do problema, quanto no desenvolvimento das aplicações que os reutilizam.

Embora existam componentes nas áreas de negócios estes muitas vezes não são bem documentados, o que dificultam suas reutilizações e manutenções. Motivado também pela necessidade de padronização de componentes de software na área de saúde, pelo conhecimento adquirido na área de saúde através do desenvolvimento de sistemas de software para área de cardiologia, e pela necessidade de cooperação do Instituto do Coração de Marília [ICM, 2004], onde este pesquisador tem um software de cardiologia desenvolvido, com o

Consórcio de Componentes de Software (CCS-SIS), este projeto adotou o domínio de cardiologia como um estudo de caso para apresentar a abordagem para construção e reutilização de componentes de software.

1.4 Organização do Trabalho

Para suportar a construção e reutilização dos componentes são utilizadas diferentes tecnologias:

- Componentes de Software;
- Padrões de Projeto;
- Engenharia de Domínio;
- Métodos de Desenvolvimento de Software Baseado em Componentes;
- Ferramentas CASE para construção dos componentes e das aplicações;
- Sistema de Transformação Draco-PUC, para geração de código *ObjectPascal*; e
- Linguagem Orientada a Componentes, *ObjectPascal* (*Delphi*), para implementação de Componentes e das Aplicações desenvolvidas.

O capítulo 2 apresenta os Principais Conceitos envolvidos na abordagem, onde a seção 2.1 apresenta uma revisão sobre Componentes de Software, a seção 2.2, Padrões de Projetos, a seção 2.3 apresenta conceitos sobre Engenharia de Domínio, a seção 2.4 apresenta conceitos e métodos sobre Desenvolvimento de Software Baseado em Componentes, a seção 2.5 apresenta as Ferramentas CASE, a seção 2.6 apresenta Sistemas de Transformação de Software, e, finalmente, a seção 2.7, a Linguagem *ObjectPascal (Delphi)*.

O capítulo 3 apresenta uma visão geral da Abordagem para Construção e Reutilização de Componentes de Software, onde são reunidos diferentes conceitos estudados.

O capítulo 4 mostra a Construção de Componentes de Software, primeira etapa da abordagem, e o Capítulo 5 mostra o desenvolvimento de aplicações, reutilizando os componentes construídos na etapa anterior.

O capítulo 6 apresenta a avaliação da aplicação da presente abordagem em um domínio específico. No capítulo 7 tem-se a conclusão do projeto apresentado, bem como as principais contribuições do trabalho desenvolvido.

Capítulo 2

Principais Conceitos

Este capítulo apresenta os principais conceitos fundamentais para o entendimento da abordagem. Estes conceitos estão relacionados com as principais tecnologias usadas pela abordagem: Componentes de Software, Padrões de Projeto, Engenharia de Domínio, Métodos de Desenvolvimento de Software Baseado em Componentes (DBC), o Sistema de Transformação de Software, a Linguagem *ObjectPascal* (*Delphi*) e Ferramentas CASE.

2.1 Componentes de Software

O conceito exato de componente em DBC ainda não é um tópico fechado. Cada grupo de pesquisa caracteriza, da maneira mais adequada ao seu contexto, o que seria um componente e, assim, não há ainda, na literatura, uma definição comum para este termo.

Pesquisas [Braga, 2000], [Werner, 2000] apontam que, desde o primeiro workshop de DBC, em 1998, em Kyoto, várias definições têm sido apresentadas. Cada uma delas ressalta um determinado aspecto de um componente. Atualmente, nota-se que já existem definições mais precisas e convergentes a respeito do que vem a ser um componente. O conceito dado por Sametinger [Sametinger, 1997] é abrangente o suficiente para estabelecer uma definição satisfatória do que seria um componente em DBC:

"Componentes reutilizáveis são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras em conformidade com um dado modelo de arquitetura de software, documentação apropriada e um grau de reutilização definido".

Ser autocontido significa que a função que o componente desempenha deve ser realizada por ele, de forma completa. Um componente deve, também, ser claramente identificável, de forma que possa ser encontrado de maneira facilitada. Este atributo,

inclusive, vem sendo alvo de inúmeras pesquisas [Werner, 1998], [Werner, 2000], e [Werner, 2000a].

Um componente não é completamente independente dos outros componentes e do ambiente. O que determina como se dá esta dependência do componente, em relação aos demais e ao ambiente que o cerca, são suas interfaces. As interfaces de um componente determinam como este componente pode ser reutilizado e interconectado com outros componentes.

Szyperski [Szyperski, 1998] define uma interface como sendo um conjunto de assinaturas de operações que podem ser invocadas por um cliente. Para cada operação, sua semântica deve ser especificada e esta especificação tem um papel duplo, que é servir tanto para os servidores que implementam a interface quanto para os clientes que irão usar esta interface.

A documentação do componente de software é também indispensável para a reutilização. Esta deve ser suficiente para que se possa recuperar um componente, avaliar sua adequabilidade para o contexto da reutilização, fazer adaptações (se for o caso) e integrar o componente ao seu novo ambiente. Outro conceito importante é o que os pesquisadores chamam de "grau de reutilização" do componente. O grau de reutilização contém informações, tais como: quantas vezes e onde o componente foi reutilizado, quem é responsável pela manutenção do componente e quem é o proprietário [Sametinger, 1997], [Braga, 2000].

Outra questão importante é que, no início, a visão de um componente como um elemento, somente como código, era bastante forte, e, com o passar dos anos [D'Souza, 1998], [Braga, 2000], [Pressman, 2001], esta visão está mudando e as pessoas começam a visualizar componentes em todos os níveis de abstração do processo de desenvolvimento.

Atualmente, existe um consenso em relação a dois grupos principais de componentes. Szyperski [Szyperski, 1998] e WALLNAU [CBSE, 2002] classificam os componentes em:

- Componentes de negócio, que são componentes que o domínio em si consegue reconhecer, tais como paciente, consulta, laudos, e outros.
- Componentes de infra-estrutura, que são componentes de suporte aos componentes de negócio, tais como segurança, distribuição, persistência em banco de dados, auditoria e outros.

Existe também um consenso em relação à importância dos componentes de negócio, apesar de os mesmos ainda não serem tão bem explorados. Esta importância se deve ao fato

de que os pesquisadores na área concordam que o que irá dar retorno, em termos de investimento, para as empresas, são os componentes que descrevem as funcionalidades do negócio da empresa [Braga, 2000].

O Departamento de Informática do SUS (DATASUS) [DATASUS, 2002] é o órgão governamental responsável por coletar, processar e disseminar informações sobre saúde, bem como padronizar o processamento destas informações, utilizando-se para isto do Consórcio de Componentes de Software (CCS-SIS), que tem por objetivo produzir interfaces de componentes especializados para aplicações de saúde especificadas por uma metodologia aberta. Adicionalmente o consórcio garantirá a implementação e disponibilização dos componentes especificados para uso dos hospitais e empresas de informática que atuam na área. Os componentes permitirão a construção de sistemas de informação mais ágeis e flexíveis, com maior facilidade de manutenção e adaptação aos diferentes cenários regionais.

Hoje já existem vários componentes em diferentes áreas, particularmente para suportar a construção de GUI *(Graphic User Interface)* como nos ambientes *Delphi*, *JBuilder* [Borland, 2002] e outros. Componentes para os domínios de aplicações são mais raros, porém existem vários componentes padronizados para a área de saúde, fruto do trabalho compartilhado de várias instituições como o Centro de Informática em Saúde da Universidade Federal de São Paulo – CIS-EPM/UNIFESP [CIS-UNIFESP, 2002], o Hospital de Clínicas da Universidade de São Paulo [HCSP-USP, 2002], as empresas de Processamento de Dados de Belo Horizonte –PRODABEL [PRODABEL, 2002] e de Porto Alegre – PROCEMPA [PROCEMPA, 2002], o Instituto do Coração do Hospital de Clínicas da Faculdade de Medicina da USP [InCor, 2002], o Hospital de Clínicas de Porto Alegre - HCPA-UFRGS [HCPA-UFRGS , 2002] e a Sociedade Brasileira de Informática em Saúde [SBIS, 2002].

2.2 Padrões de Projeto

A utilização de padrões no desenvolvimento de software tem emergido como uma das mais promissoras abordagens para a melhoria da qualidade de software.

O conceito de padrões foi proposto pelo arquiteto Alexander [Alexander, 1977] para construção de arquiteturas e planejamento urbano. Na área de software, foram recentemente

popularizados para apoiar uma larga variedade de domínios relacionados às atividades de desenvolvimento de projetos.

A utilização de padrões em sistemas complexos de softwares permite que soluções, previamente testadas, sejam reutilizadas, tornando o sistema mais compreensível, flexível, fácil de desenvolver e manter. O objetivo do uso de padrões de software é a disseminação das soluções de desenvolvimento de software já existentes [Johnson, 1992].

Padrões são soluções de um problema previamente testadas e aprovadas. Os desenvolvedores baseiam-se nesses padrões para melhor resolverem seus problemas, aumentado o grau de reutilização de uma aplicação [Fontes, 2001, Sanches, 2000]. Alexander [Alexander, 1977] descreve que um padrão é constituído de três partes: um *Contexto*, um *Problema*, e uma *Solução*.

O *Contexto* descreve o domínio onde o problema ocorre; o *Problema* é uma questão que ocorre repetidamente e que deve ser solucionada e a *Solução* mostra como resolver o problema.

Buschmann [Buschmann , 1995] divide os padrões em três categorias que representam diferentes níveis de abstração:

- *Padrões Arquiteturais (Architectural Patterns):* descrevem um esquema de organização estrutural e fundamental para um sistema de software. Eles fornecem um conjunto de subsistemas pré-definidos, especificando as responsabilidades e definindo regras e diretrizes para organizar os relacionamentos entre eles;
- *Padrões de Projeto (Design Patterns):* fornecem um esquema para refinar os subsistemas, os componentes de sistemas de software ou o relacionamento entre eles. Os mecanismos de cooperação entre componentes são descritos e definidos para encontrar soluções para os problemas de projeto em um contexto específico. Gamma [Gamma, 1995] define que um padrão de projeto nomeia, abstrai e identifica o essencial da estrutura de um projeto comum, tornando-o útil para criação de um projeto Orientado a Objeto reusável; e
- *Padrões em nível de Idiomas (Idioms):* especificam como implementar os aspectos particulares dos componentes e seus relacionamentos adequadamente às características da linguagem de programação.

Um Padrão de Projeto descreve uma estrutura comum para criar um projeto orientado a objetos reutilizáveis, baseados em soluções práticas. O termo padrão de projeto refere-se freqüentemente a qualquer padrão que diretamente trata de arquitetura de software, de projeto ou de implementação.

Buschmann [Buschmann, 1995] também divide os padrões em três grupos que representam o grau de relacionamento entre os padrões:

Catálogo de Padrões (Patterns Catalog): descrevem vários padrões, cada um de uma forma relativamente independente, para problemas comuns de projeto;

Sistemas de Padrões (Patterns Systems): descrevem os relacionamentos entre os padrões e como estão conectados. Alexander [Alexander, 1977] usa o termo Linguagem (*language*) em comparação com Sistema; e

Linguagens de Padrões (Patterns Languages): descrevem soluções para todos os problemas de projeto que podem ocorrer em um domínio.

Enquanto um Catálogo de Padrões representa um conjunto de padrões relacionados por um número pequeno de categorias, um Sistema de Padrões representa um conjunto de padrões organizados em grupos e subgrupos para apoiar a construção e evolução de arquiteturas. Já as Linguagens de Padrões definem um conjunto de padrões associados, formando uma estrutura organizacional, produzindo arquiteturas completas. Linguagens de Padrões evoluem a partir de sistemas de padrões, por meio de um processo de implantação de novos padrões e de regras para o relacionamento entre os mesmos.

Gamma [Gamma, 1995] propõe o uso de padrões de projeto de software como um novo mecanismo para expressar soluções na elaboração de projetos orientados a objetos. Padrões de projeto fornecem uma linguagem comum, que irá facilitar a comunicação entre desenvolvedores e projetistas. Melhoram o aprendizado de jovens desenvolvedores, incrementando a padronização do desenvolvimento, permitindo a construção de softwares reutilizáveis, que se comportam como blocos de construção para sistemas mais complexos. Gamma apresenta os conceitos básicos sobre os padrões, as formas para selecioná-los e utilizá-los. Como estudo de caso, um editor de texto, denominado Lexi, foi desenvolvido para solucionar vários problemas de projeto existentes, com o uso dos padrões. Seus autores respondem de forma prática às questões do tipo: como reconhecer um problema e como escolher e utilizar um padrão de projeto. É também apresentado um catálogo de padrões de projeto, organizado segundo dois critérios:

a) **propósito** - indica o que o padrão de projeto faz;
b) **escopo** - especifica se o padrão de projeto é aplicável a classes ou objetos.

Quanto ao propósito, os padrões de projeto são divididos em três categorias: criacionais, estruturais e comportamentais. Padrões de projeto criacionais organizam o processo de criação de objetos. Padrões de projeto estruturais demonstram a composição de

classes ou objetos. Padrões de projeto comportamentais caracterizam as formas como os objetos interagem entre si.

Em geral, um padrão possui quatro elementos essenciais [Gamma, 1995], [Prieto-Diaz, 1990]:

- **nome**: uma designação que descreve com propriedade um problema de projeto, suas soluções e conseqüências em uma palavra ou duas;
- **problema:** descreve o problema e o contexto em que o mesmo ocorre. Pode conter um conjunto de condições que devem ser satisfeitas para a aplicação do padrão de projeto;
- **solução:** descreve os elementos (classes e objetos) necessários para a construção do projeto, seus relacionamentos, responsabilidades e colaborações. Essa solução não é restrita a um caso em particular, e sim genérica o suficiente para ser aplicada a diferentes situações;
- **conseqüências:** são os resultados, positivos e negativos, ao se utilizar padrões de projeto. As conseqüências permitem analisar os custos e benefícios da aplicação de um determinado padrão de projeto.

No projeto do Desenvolvimento de Software Baseado em Componentes serão utilizados padrões do ambiente *Delphi*, apresentados a seguir.

2.2.1 Padrões de Projeto no Ambiente *Delphi*

Delphi implementa uma linguagem inteiramente orientada a objetos com muitos refinamentos práticos que simplificam o desenvolvimento.

As características mais importantes de uma classe da perspectiva de um padrão, são as heranças das classes, os métodos virtuais (*virtual*) e abstratos (*abstract*), e o uso dos escopos protegido (*protected*) e público (*public*). Estas características dão as ferramentas para criar os padrões, que podem ser reutilizados e estendidos e permitem isolar a funcionalidade, variando os atributos-base que não são alterados.

Delphi é um grande exemplo de uma aplicação com extensibilidade, através de sua arquitetura de componentes, das interfaces *IDE* e ferramentas para geração de interfaces. Estas interfaces definem muitos construtores e operações virtuais e abstratas.

A Tabela 1 apresenta os principais padrões de software, com uma definição resumida de seu objetivo. Estes padrões serão utilizados com mais freqüência na construção dos componentes.

Tabela 1 - Padrões de Projeto

Nome do Padrão	Definição
Singleton	"Assegura que determinada classe tenha apenas uma instância e providencia uma forma de acesso global à mesma".
Adapter	"Facilita a conversão da interface de uma classe para outra interface mais interessante para o cliente, fazendo com que várias classes possam trabalhar em conjunto independentemente das interfaces originais".
Template Method	"Define o corpo de um algoritmo em uma operação, deixando alguns passos para as subclasses. O padrão *Template Method* permite que as subclasses redefinam determinadas etapas de um algoritmo sem mudar a estrutura do algoritmo".
Builder	"Separa a construção de um objeto complexo de sua representação de modo que o mesmo processo da construção possa criar representações diferentes".
Abstract Factory	"Fornece uma interface para criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas".
Factory Method	"Define interfaces para criar um objeto, mas deixa as subclasses decidirem que classe instanciar. *Factory Method* deixa a classe ser instanciada pela subclasse".

Segue-se a apresentação mais detalhada dos padrões da Tabela 1, com suas implementações em *ObjectPascal* (*Delphi*).

2.2.1.1 Singleton

O padrão *Singleton* [Gamma, 1995], [Larman, 1999] assegura que determinada classe tenha apenas uma instância e providencia uma forma de acesso global à mesma. Todos os objetos que desejam utilizar-se de um objeto dessa classe irão apenas compartilhar a mesma instância. Na maioria dos casos, esta classe está associada ao acesso e gerenciamento de um determinado recurso ou serviço.

A proposta desse padrão é transferir para a classe a responsabilidade pela criação e administração dessa instância única. O *Singleton* deve assegurar que nenhuma outra instância possa ser criada e deve providenciar um mecanismo de acesso a sua instância. A Figura 1 [Gamma, 1995] apresenta o diagrama de classes desse padrão.

O construtor da classe *Singleton* deve ser do tipo privado. Isto impossibilita a criação direta de uma instância por outras classes. Toda vez que for necessário acessar a classe *Singleton*, deve-se utilizar o método `getInstance()`.

O método `getInstance()` é um método estático, que sempre retorna o mesmo objeto da classe *Singleton.* A instância retornada é referenciada pela variável estática `singletonInstance`, que é do tipo privado, e a sua instanciação ocorre somente no momento em que a classe é criada e alocada na memória [Gamma, 1995], [Prieto-Diaz, 1990].

Há diversos exemplos deste método na biblioteca de componentes visuais do *Delphi* (VCL), como as classes TApplication, TScreen or TClipboard.

O padrão é útil sempre que se quer um único objeto global em sua aplicação. Outros usos podem incluir um alimentador global da exceção, a segurança da aplicação, ou um único ponto da interface de uma outra aplicação.

Para implementar uma classe deste tipo, use *override* nos métodos *constructor* e o *destructor* da classe para referenciar a uma variável global (interface) de uma classe.

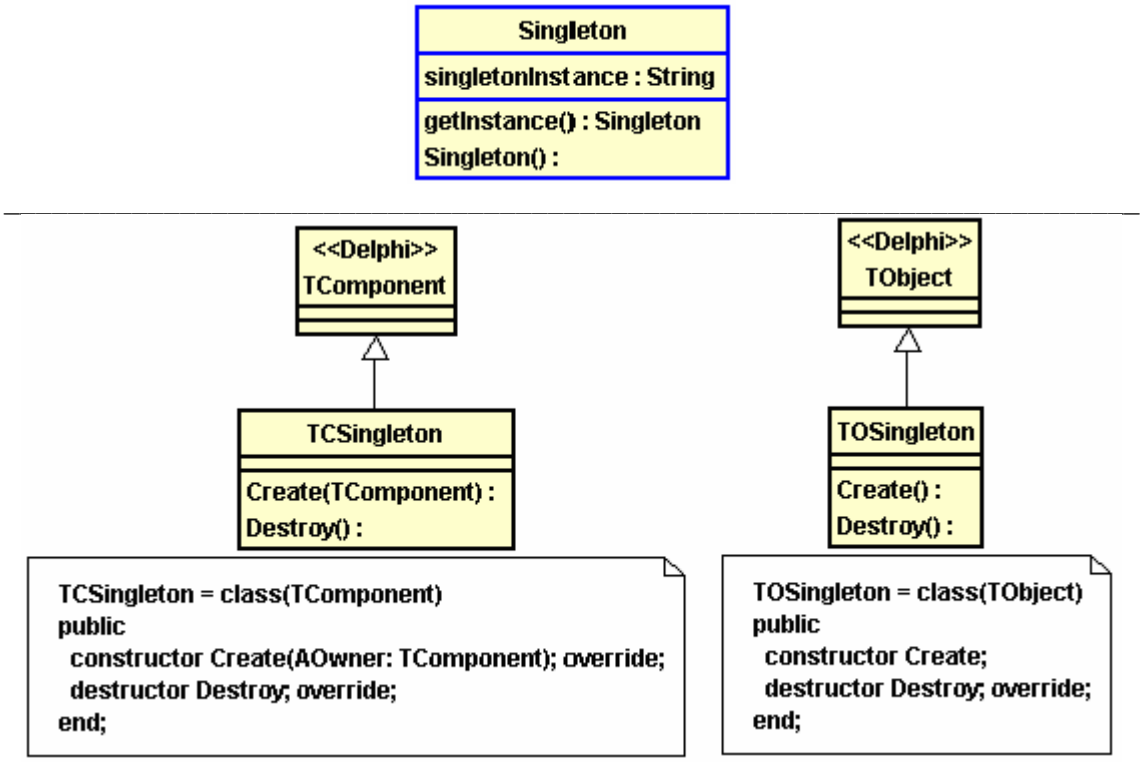


Figura 1 – Diagrama de Classes do Padrão *Singleton*

2.2.1.2 Adapter

Um exemplo típico deste padrão no ambiente *Delphi* é gerado quando se importa um OCX. *Delphi* gera uma nova classe a qual disponibiliza uma interface para um controle externo dentro de uma interface compatível para o Pascal. Um outro caso típico é quando o usuário quer construir uma interface simples para um antigo ou um novo sistema.

O *Delphi* não permite a adaptação de classes através de herança múltipla. O *Adapter* necessita fazer uma referência para uma instância específica de uma classe antiga.

O exemplo que se segue na Figura 2 é uma implementação que ilustra o tratamento que pode ser dado à questão do problema do ano 2000. Tem-se uma classe **TNewCustomer** (que trata o ano com 4 dígitos) e uma classe **TOldCustomer** (que trata o ano com 2 dígitos). A nova classe, **TNewCustomer**, tem o atributo ***FDOB*** que trata o ano com 4 dígitos, as operações protegidas são visíveis somente nas classes descendentes e as operações e as propriedades possuem métodos de leitura e escrita, podendo reescrever a operação sem que a classe pai saiba sobre sua nova implementação.

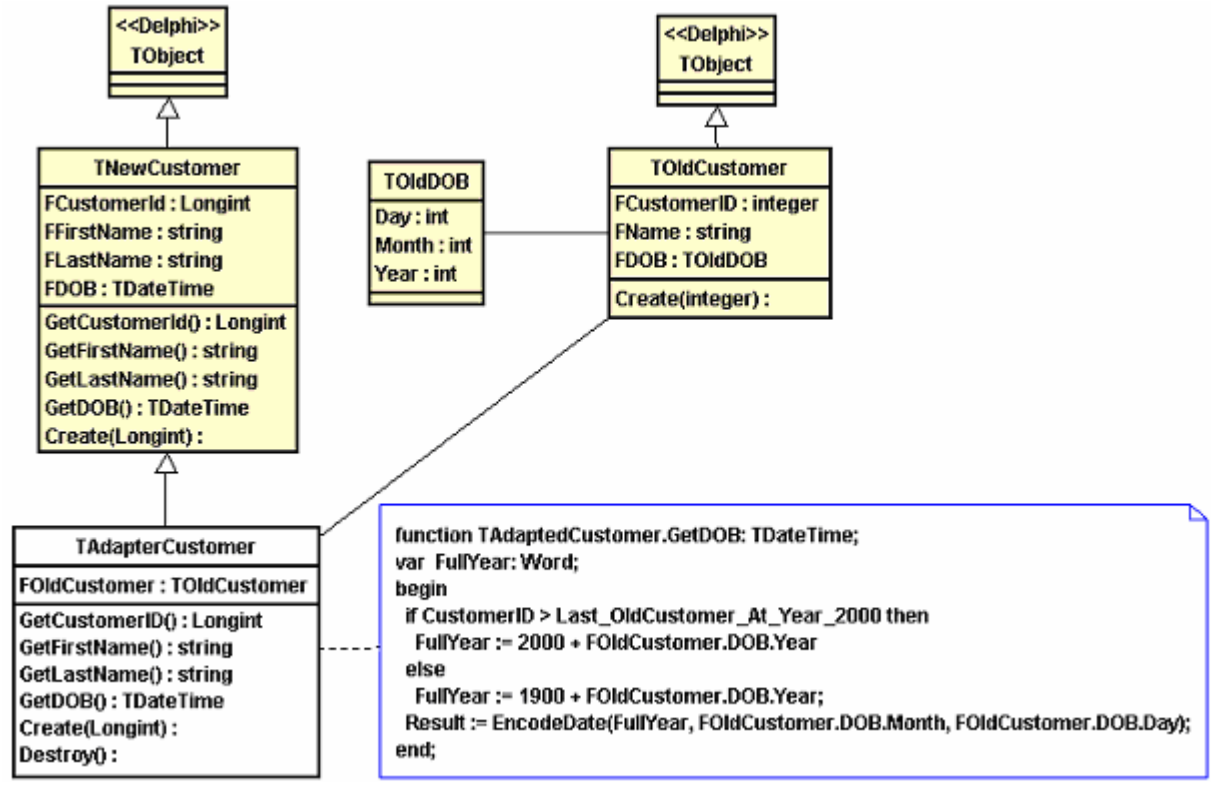


Figura 2 - Padrão *Adapter*

2.2.1.3 Template Method

Abstração é implementada no ambiente *Delphi* por métodos virtuais abstratos. Métodos Abstratos (*Abstract*) diferem de métodos Virtuais (*Virtual*) pela classe base por não possuir qualquer tipo de implementação. A classe descendente é responsável pela implementação de um método abstrato. Chamando um método abstrato que não tenha utilizado *override* resultará em um erro.

Um exemplo típico de abstração é a classe **TGraphic**.

TGraphic é uma classe abstrata usada para implementar **TBitmap**, **TIcon** e **TMetafile**. Outros desenvolvedores usam **TGraphic** como uma classe base para outros objetos gráficos como as extensões PCX, GIF, JPG. **TGraphic** define métodos abstratos como ***Draw***, ***LoadFromFile*** and ***SaveToFile***, os quais são então reescritos em classes concretas. Outros objetos que usam **TGraphic**, como **TCanvas**, somente conhecem o método abstrato ***Draw***, ainda que sejam utilizados como classes concretas em tempo de execução.

A Figura 3 mostra os passos de um algoritmo que ilustra os princípios de atribuição de implementação para uma subclasse.

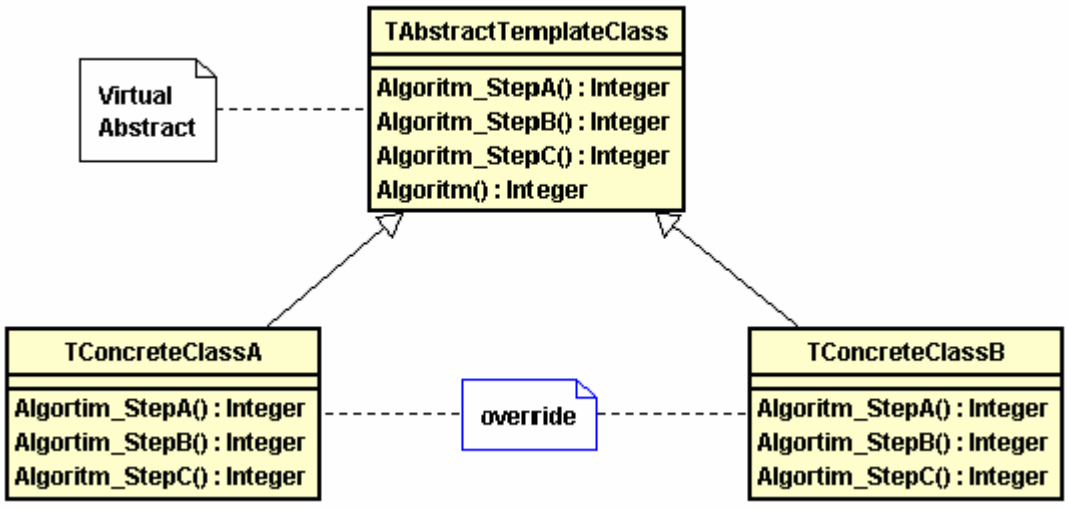


Figura 3 - Padrão *Template Methods*

Para implementar *Template Methods* é necessária uma classe abstrata e uma classe concreta para cada alternativa de implementação. Define-se uma interface pública (*public*) para um algoritmo em uma classe base abstrata. No método público (*public*), são

implementados os passos do algoritmo chamados por métodos abstratos protegidos (*protected abstract methods*) da classe.

Na classe concreta, derivada da classe base, implementa-se (usando *override*) cada passo do algoritmo com uma implementação específica da classe concreta.

2.2.1.4 Builder

O Padrão *Builder* possui conceito semelhante ao *Abstract Factory*. A diferença é que é o *Builder* se permite criar os objetos de classes concretas diferentes, mas que contém as várias partes, visto que o *Abstract Factory* permite criar famílias inteiras de classes concretas.

A funcionalidade utilizada na VCL do *Delphi* para criar *Forms* e componentes é similar ao conceito para o *Builder*. O *Delphi* cria *Forms* usando uma interface comum, através do método *Application.CreateForm* e o construtor da classe *TForm*. *TForm* implementa um construtor comum usando informações do fonte (DFM) para instanciar os componentes próprios do *Form*.

Muitas classes descendentes reutilizam o mesmo processo de construção para criar diferentes representações. O evento *OnCreate* (*TForm*) também adiciona um gancho dentro do processo construtor para permitir estender a funcionalidade mais facilmente.

O exemplo a seguir, na Figura 4, inclui a classe ***TAbstractFormBuilder*** e duas classes concretas ***TRedFormBuilder*** and ***TBlueFormBuilder***.

Em tempo de execução as aplicações clientes instruem as classes concretas para criar as partes usando os métodos construtores declarados como *public*. A instância concreta do *Builder* é passada para o seguinte método:

```
procedure TForm1.Create3ComponentFormUsingBuilder(ABuilder:
TAbstractFormBuilder);
var
  NewForm: TForm;
begin
  with ABuilder do begin
    CreateForm(Application);
    CreateEdit;
    CreateSpeedButton;
    CreateLabel;
    NewForm := Form;
    if NewForm <> nil then NewForm.Show;
  end;
end;
```

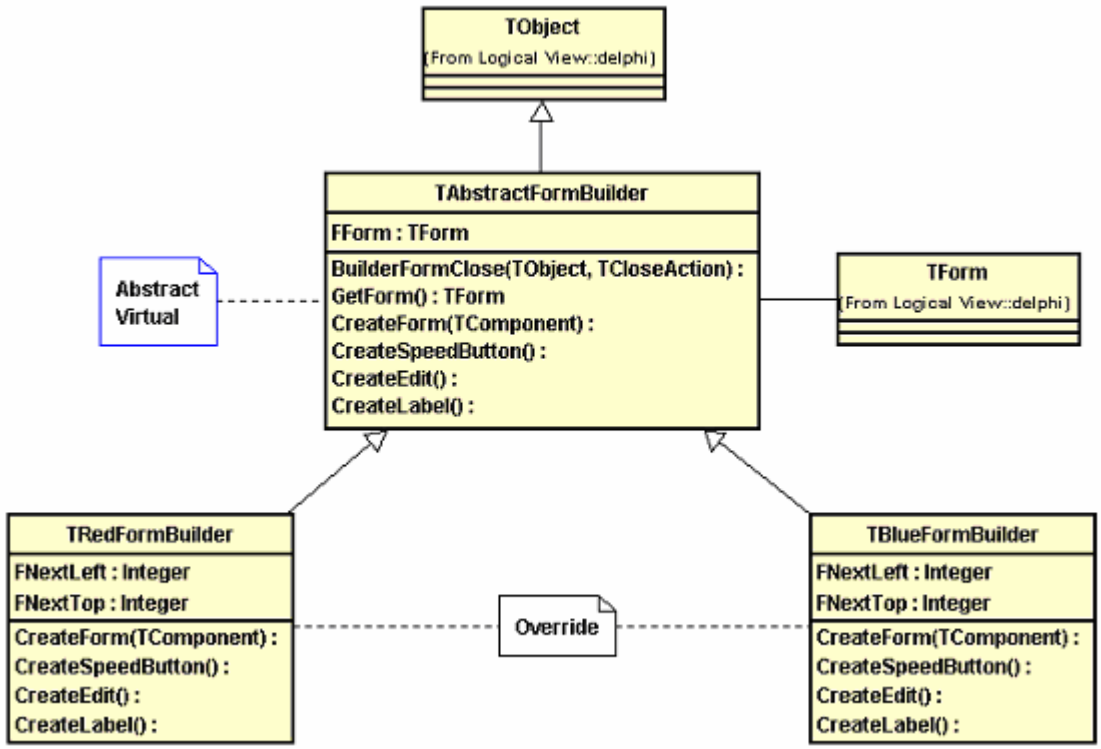


Figura 4 - Padrão *Builder*

2.2.1.5 Abstract Factory

O padrão *Abstract Factory* fornece uma interface para criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas [Gamma, 1995], [Larman, 1999]. O padrão *Abstract Factory* é aplicado quando é necessário fornecer uma biblioteca de classes de produtos somente com suas interfaces, implementando-as em classes concretas.

Este padrão é ideal quando se quer isolar a aplicação de uma implementação de uma classe concreta. Por exemplo, se for necessário cobrir a VCL do *Delphi* com camadas comuns VCL para aplicações de 16 e 32 bits, deve-se iniciar com o padrão *Abstract Factory* como base.

A Figura 5 mostra o padrão *Abstract Factory* e duas classes *Concrete Factory* para implementar diferentes estilos de componentes para interface do usuário. *TOAbstractFactory* é uma classe *singleton*.

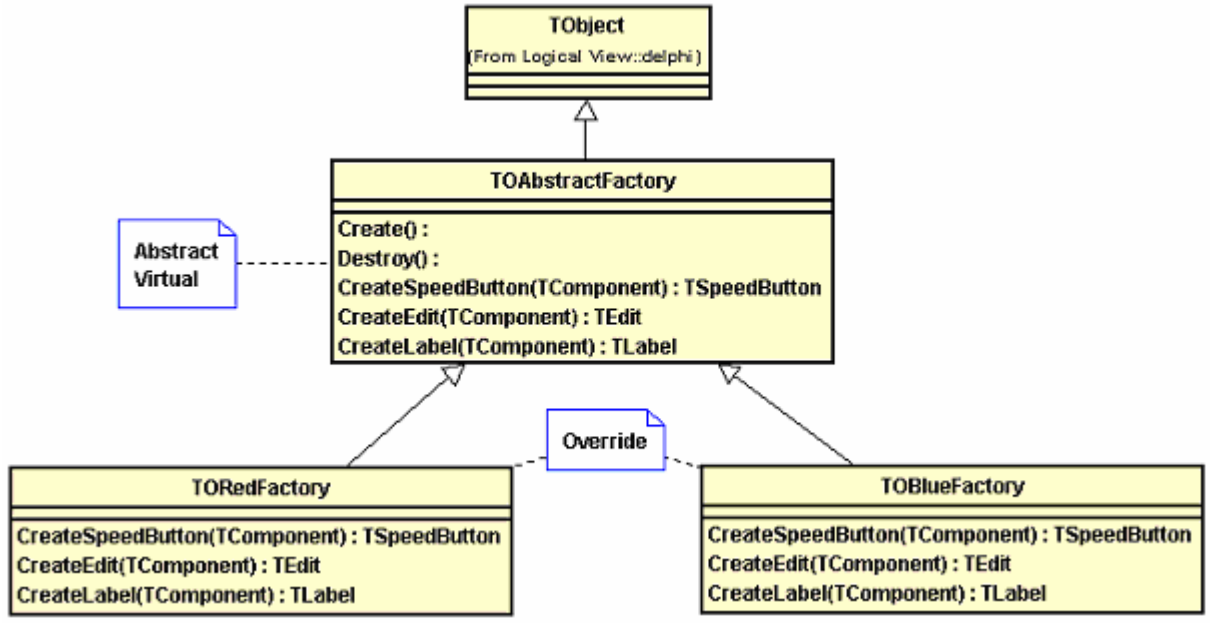


Figura 5 - Padrão *Abstract Factory*

TORedFactory e ***TOBlueFactory*** usam *override* para a interface abstrata para suportar diferentes estilos.

2.2.1.6 Factory Method

O padrão *Factory Method* define interfaces para criar um objeto, mas deixa as subclasses decidirem que classe instanciar. O padrão ainda permite adiar a instanciação para subclasses [Gamma, 1995], [Larman, 1999]. Este padrão é utilizado quando se pretende encapsular a construção de uma classe e isolar o conhecimento de uma classe concreta de uma aplicação cliente através de uma interface abstrata.

Um exemplo é uma aplicação de negócio, orientada a objeto, com opções para se conectar a vários SGBD. A aplicação cliente somente precisa saber sobre as classes de negócios, e não sobre suas implementações para armazenamento e recuperação.

No exemplo do padrão *Abstract Factory*, cada função do construtor virtual é um *Factory Method*. Em suas implementações são definidas as especificações da classe para o retorno. A Figura 6 mostra um exemplo do Padrão *Factory Method*.

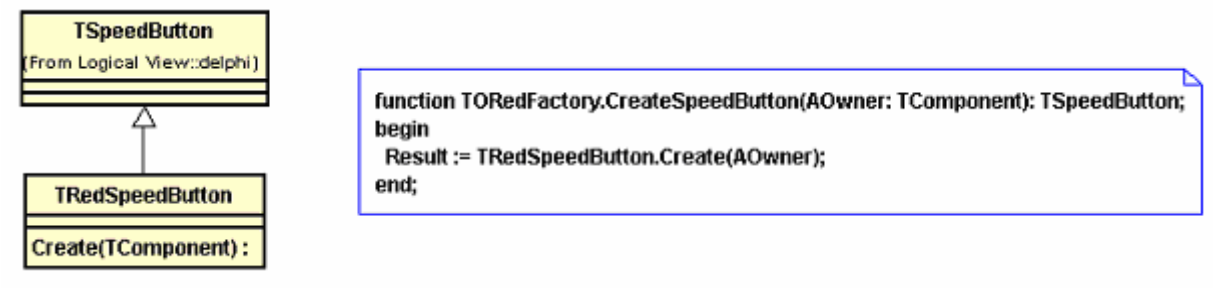


Figura 6 - Padrão *Factory Method*

2.3 Engenharia de Domínio (ED)

Segundo Pressman [2001], a intenção da Engenharia de Domínio (ED) é identificar, construir, catalogar e disseminar um conjunto de componentes de software que possua aplicabilidade para software existente ou futuro, dentro de um particular domínio de aplicações. Deste modo, a ED deve estabelecer mecanismos que sistematizem a busca e a representação de informações do domínio, de forma a facilitar ao máximo a sua reutilização [Werner, 2000].

2.3.1 Atividades da Engenharia de Domínio

Pesquisas [Prieto-Diaz, 1990], [Griss, 1998], [Pressman, 2001] apontam que existem basicamente três etapas principais na ED:

- ***Análise do Domínio***: corresponde ao processo pelo qual a informação usada no desenvolvimento do sistema, para um determinado domínio, é identificada, capturada e organizada, com a proposta de ser reutilizada para a criação de novos sistemas [Prieto-Diaz, 1990];
- ***Projeto do Domínio***: nesta etapa, os resultados da análise do domínio são usados para identificar e generalizar soluções para os requisitos comuns, através da especificação de uma arquitetura de software do domínio. As oportunidades de reutilização, identificadas na análise do domínio, são refinadas de forma a especificar as restrições do projeto [Werner, 2000]; e
- ***Implementação do Domínio***: nesta etapa, transformam-se as oportunidades de reutilização e soluções do projeto para um modelo implementacional, que inclui

serviços, tais como: a identificação, reengenharia e/ou construção, e manutenção dos componentes reutilizáveis, que suportem estes requisitos e soluções de projeto [Werner, 2000].

Para se ter uma idéia do que é realmente a Análise de Domínio (AD), é importante se definir primeiramente o seu objeto de estudo: o *domínio do problem*a. Segundo Shapere [Shapere, 1977], em uma certa comunidade, informações sobre o mundo real podem ser associadas em grupos de informações ou domínios de problema, de forma que: relacionamentos compreensíveis existentes entre os itens de informação sejam identificados com respeito a alguma classe de problemas; os problemas possuam significado para os membros da comunidade; e exista o conhecimento necessário à solução do problema.

Esta definição geral cobre duas perspectivas: domínio como uma coleção de problemas reais e domínio como uma coleção de aplicações (existentes ou futuras) que compartilhem características comuns. Esta distinção afeta, profundamente, a maneira como será realizada a AD. Para o primeiro ponto de vista, o resultado da AD deve ser uma teoria sobre os problemas do domínio. Esta teoria deve ser capaz de auxiliar a forma de pensar sobre o domínio, ou seja, predizer, explicar ou derivar fatos sobre o domínio que seriam dificilmente observados diretamente. Para o segundo ponto de vista, o resultado da AD é uma taxonomia de sistemas que demonstre as similaridades e diferenças entre os sistemas estudados, organizando os componentes e arquiteturas comuns a estes [Arango, 1994].

2.3.2 Profissionais envolvidos com a Engenharia de Domínio

Em [Simos, 1996], [Werner, 2000], são apresentados três grupos principais de profissionais que atuam de forma efetiva num processo de ED:

- ***Fontes***: são os usuários finais, que utilizam aplicações já desenvolvidas naquele domínio, e os especialistas do domínio, que provêem informações a respeito de conceitos e funcionalidades relevantes no domínio;
- ***Produtores***: são os analistas e os projetistas do domínio, que capturam as informações das fontes e de aplicações existentes e realizam a análise, o projeto, a implementação e o empacotamento dos componentes reutilizáveis; e
- ***Consumidores***: são os desenvolvedores de aplicações e usuários finais interessados no entendimento do domínio, que utilizam os modelos gerados nas

etapas anteriores para a especificação de aplicações e/ou maior entendimento dos conceitos e funções inerentes ao domínio.

2.3.3 Produtos gerados pela Engenharia de Domínio

Um método de engenharia de domínio, para ser efetivo, deve prover representações específicas para documentar os resultados de cada uma das fases da ED [Werner, 2000]. Estas representações devem, prioritariamente, representar o escopo e a abrangência do domínio, descrever os problemas passíveis de serem solucionados, através de componentes de software, no domínio, e prover especificações arquiteturais e implementacionais que solucionem os problemas encontrados do domínio. Desta forma, para cada uma das fases da engenharia de domínio, devem existir produtos específicos a serem disponibilizados.

Na fase de **análise de domínio**, devem ser disponibilizadas representações que capturem o contexto e a abrangência no domínio, explicitando seu relacionamento com outros domínios. Nesta fase, ainda, devem ser disponibilizados modelos que capturem os principais conceitos e funcionalidades inerentes ao domínio, gerando, assim, um ou mais modelos com abstrações do domínio [SEI, 1999a].

O **projeto do domínio** deve disponibilizar modelos que especifiquem a estrutura arquitetural a ser seguida pelas aplicações do domínio. As representações geradas devem prover modelos arquiteturais que auxiliem na especificação de arquiteturas específicas para cada aplicação [Werner, 2000].

A fase de **implementação do domínio** deve disponibilizar componentes implementacionais que especifiquem as principais funcionalidades encontradas em aplicações do domínio. Estes componentes implementacionais devem estar em conformidade com o modelo de abstrações da fase de análise e com os modelos arquiteturais da fase de projeto, de forma que possam cooperar entre si para implementar todas as funcionalidades necessárias para uma determinada aplicação [SEI, 1999b].

2.4 Desenvolvimento de Software Baseado em Componentes (DBC)

Até bem pouco tempo atrás, o desenvolvimento da maioria dos produtos de software disponíveis no mercado era baseado em uma abordagem de desenvolvimento em blocos monolíticos, formados por um grande número de partes inter-relacionadas, cujos relacionamentos estavam na maioria das vezes implícitos. O DBC surgiu como uma nova perspectiva para o desenvolvimento de software, cujo objetivo é a "quebra" destes blocos monolíticos em componentes interoperáveis, reduzindo desta forma a complexidade no desenvolvimento, assim como os custos, através da utilização de componentes que, a princípio, seriam adequados para serem utilizados em outras aplicações [Sametinger, 1997].

Como todo processo de desenvolvimento de software, é preciso prover uma sistemática para o desenvolvimento baseado em componentes, devendo ser cuidadosamente planejada e controlada para ser efetiva. Assim, o Processo de Desenvolvimento de Software (PDS) deve se preocupar com a reutilização em todas as etapas do desenvolvimento, não apenas com o código.

Segundo Krueger [Krueger, 1992], o processo de desenvolvimento pode ser dividido em duas etapas: desenvolvimento **para** reutilização e desenvolvimento **com** reutilização. Para uma técnica de reutilização de software (como é o caso do desenvolvimento baseado em componentes) ser efetiva, ela deve reduzir a distância cognitiva entre o conceito inicial de um sistema e sua implementação final. Sendo assim, no contexto de um processo de desenvolvimento de software baseado em componentes, a aplicação da reutilização nas fases iniciais do desenvolvimento deve ter como resultado a reutilização de componentes na implementação da aplicação, ou seja, deve haver uma relação estreita entre os conceitos de um dado domínio de aplicação considerados reutilizáveis nas fases iniciais do processo de desenvolvimento e os componentes codificados que serão utilizados na implementação da aplicação.

Considerando estas idéias no contexto do desenvolvimento baseado em componentes, esta nomenclatura poderia ser mudada e teríamos a mesma divisão: desenvolvimento **de** componentes e desenvolvimento **com** componentes [Braga, 2000].

Ao considerar componentes muito mais do que somente código, uma abordagem de desenvolvimento de componentes pode ser vista como um processo de engenharia de domínio completo, onde a preocupação, além de ser com a disponibilização de componentes em todos os níveis de abstração, não é com uma única aplicação, mas com uma família de aplicações. Assim, o grau de reutilização de um dado componente será muito maior e seu entendimento também, uma vez que este componente será disponibilizado tanto para especificação quanto

para código. Por outro lado, o desenvolvimento **com** componentes deve ser visto como um processo de desenvolvimento de aplicações, onde a reutilização de componentes deve estar presente em todas as fases.

O desenvolvimento baseado em componentes [Szyperski, 1998] está cada vez mais ganhando adeptos e é uma das propostas promissoras para a melhoria da produção de software nos próximos anos. A construção de software através de componentes leva a uma maior reutilização e conseqüente aumento na qualidade e produtividade, o que enfatiza a crença que esta nova modalidade de desenvolvimento de software venha a crescer.

Hoje, o que temos na maioria dos produtos de software disponíveis no mercado é uma abordagem de desenvolvimento em blocos monolíticos, formados por um grande número de partes inter-relacionadas, onde o relacionamento entre estas partes não fica evidenciado através de uma interface bem definida. O desenvolvimento baseado em componentes tem como objetivo a definição de componentes interoperáveis, com interfaces bem definidas, evidenciando os tipos de relacionamentos permitidos por este componente. Desta forma, a complexidade no desenvolvimento é reduzida, assim como os custos, através da reutilização de componentes exaustivamente testados.

2.4.1 DBC e a Orientação a Objetos (OO)

Um grande foco de debate, ainda hoje, em relação ao DBC, é a sua relação com a orientação a objetos (OO) [D'Souza, 1998], [Kobryn, 2000]. Do ponto de vista do desenvolvimento OO, podemos considerar o DBC como uma evolução do mesmo. Em particular, a evolução se deve, principalmente, em termos de como se dá a construção de novos artefatos. No contexto do paradigma OO, quando um novo comportamento precisa ser definido, este é geralmente criado através da reutilização, ou herança, de um comportamento já existente no sistema, que é então especializado. Em muitas situações, esta abordagem é benéfica. No entanto, podemos observar que a mesma tem algumas limitações. Por exemplo, para sistemas complexos, as hierarquias de comportamento herdado podem se tornar estruturas pesadas, difíceis de serem entendidas e cheias de interdependências, que as tornam difíceis de serem modificadas. Assim, poderemos considerar o DBC como uma evolução da OO no sentido de pregar construções onde as hierarquias de comportamento somente sejam

permitidas no contexto de um componente. Este tipo de dependência não deve existir entre componentes, enfatizando-se, neste caso, o uso de agregações, ao invés da herança [Braga, 2000].

A tecnologia de objetos é bastante adequada para DBC, uma vez que os conceitos importantes em DBC já são inerentes ao desenvolvimento OO, tais como encapsulamento e polimorfismo. Um dos pontos conflitantes entre as duas tecnologias é justamente o uso ou não da herança. DBC, em geral, prega o não uso da herança da forma como ela é usada em OO. De acordo com Heineman [Heineman, 1999], existem dois tipos de herança: essencial, que é a herança de um comportamento ou de uma característica externa visível, e herança acidental, que é a herança de parte ou toda a implementação de um objeto mais genérico.

No desenvolvimento OO, a herança acidental feita estritamente para herança de código leva a um projeto pobre. Outro problema, ressaltado por Heineman e, também, por Szyperski, é que a utilização indiscriminada do mecanismo de herança leva a uma hierarquia de classes muitas vezes complexas, o que dificulta seu entendimento. Assim, se for necessária alguma modificação, o engenheiro de software tem que entender toda a hierarquia. Como a dificuldade deste entendimento é inerente, muitas vezes, ao invés de modificar uma dada classe, prefere-se adicionar uma nova classe, o que leva à herança acidental e ao projeto pobre [Werner, 2000].

No caso de DBC, isso é ainda mais grave, pois um dado componente deve ser conhecido apenas pelas suas interfaces externas. Assim, qualquer adaptação necessária deve ser feita idealmente através destas interfaces. Se houver a necessidade de utilizar herança, será a do tipo essencial, ou seja, herança de interfaces [Werner, 2000].

Para tentar superar essas dificuldades, o DBC conceitua que o comportamento e as interfaces dos componentes devem ser claramente especificados, sem ter que se preocupar com estruturas herdadas e suas possíveis modificações. Além disso, são providos mecanismos que permitem a conexão dos componentes de forma flexível, de tal forma que um componente possa ser substituído por outro similar a qualquer momento, sem haver a necessidade de modificações internas na estrutura dos componentes conectados.

Apesar de o DBC poder ser considerado uma evolução do desenvolvimento OO, isso não quer dizer que um componente tenha que, necessariamente, ser desenvolvido a partir da tecnologia OO. Um componente pode conter procedimentos tradicionais, ou pode ser desenvolvido inteiramente utilizando uma abordagem de desenvolvimento estruturado, ou outra qualquer. O que importa para que o mesmo seja caracterizado como sendo um componente é que esteja em conformidade com a definição de componentes em relação às

suas interfaces, arquitetura e funcionalidade, citando somente os principais aspectos [Szyperski, 1998], [Braga, 2000].

Segue-se a apresentação dos métodos *Catalysis* [D'Souza, 1999] e o *UML Components* [Cheesman, 2000].

2.4.2 Método Catalysis

Catalysis é um método de desenvolvimento de software baseado em componentes que integra técnicas, Padrões e *Frameworks*. *Catalysis* começou em 1991 como uma formalização do OMT e teve influências dos métodos *Fusion* [Coleman, 1994] e UML [Fowler, 1999], [Booch, 1994], [Booch, 1999], [Booch, 2000]. Suporta as características das tecnologias Orientadas a Objetos recentes como Java, CORBA e DCOM e sua notação é baseada na *Unified Modeling Language* (UML).

Fundamenta-se nos princípios de ***Abstração, Precisão*** e ***Componentes Plug-In.*** O princípio ***Abstração*** orienta o Engenheiro de Software na busca dos aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do sistema. O princípio ***Precisão*** objetiva descobrir erros e inconsistências na modelagem e o princípio ***Componentes "Plug-In"*** visa a reutilização de componentes para construir outros componentes.

Catalysis baseia-se em três conceitos de modelagem: ***Tipo, Colaboração e Refinamento***. *Frameworks* são usados para descrever padrões que ocorrem nestes três modelos. Com estes conceitos constrói-se uma grande variedade de padrões de modelos e projetos. Tipo e Refinamento são comuns na modelagem com precisão.

Catalysis utiliza o framework para capturar a essência de padrões. Um framework é descrito como um pacote genérico. É reutilizado substituindo apropriadamente elementos de um modelo genérico por elementos de um problema específico. O conjunto de relacionamentos, restrições e transformações de projetos em outros projetos, é chamado de Framework de Modelo.

O PDS em *Catalysis* segue as características do modelo Espiral da Engenharia de Software [Pressman, 2001], e está dividido em três níveis, conforme mostra a Figura 7: ***Domínio do Problema, Especificação dos Componentes e Projeto Interno dos Componentes***, correspondendo às atividades tradicionais do ciclo de vida do software: ***Planejamento, Especificação, Projeto e Implementação***, que são executadas de forma

incremental e evolutiva, resultando na geração de uma nova versão de um protótipo a cada ciclo realizado.

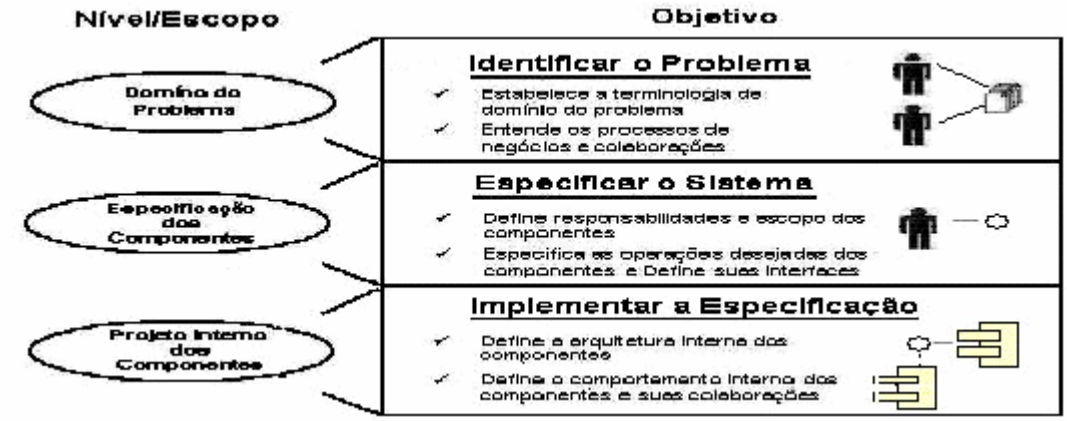


Figura 7 - Níveis de Catalysis

No nível ***Domínio do Problema*** é dado ênfase na identificação dos requisitos do sistema, especificando "o que" o sistema deve fazer para solucionar o problema. Identificam-se os tipos de objetos e ações, agrupando-os em diferentes visões por áreas de negócio.

Este nível utiliza várias tecnologias para documentá-lo, destacando-se [D'Souza, 1999]:

- *Storyboards:* modelo com as representações de diferentes situações e cenários no domínio do problema;

- *Mind-maps*: modelo com as representações estruturadas dos termos importantes do domínio do problema, relatados na entrevista pelos usuários;

- *Modelo de Colaborações:* modelo de *use-case* que descreve as ações (*use-case)* do sistema atual "como é" (*as-is)* e do sistema proposto "como será" (*as-be*), em vários níveis de abstração; e

- *Diagrama de Contexto:* modelo que define as fronteiras, atores e ações externas que estimulam o sistema;

No nível ***Especificação dos Componentes*** dá-se ênfase na identificação, comportamento e responsabilidades dos componentes. Neste nível faz-se o refinamento dos modelos de negócios, do domínio do problema, e utilizam-se várias tecnologias para documentá-lo, entre elas, pode-se utilizar:

- *Modelo de Tipos:* modelo que especifica o comportamento dos objetos, mostrando os atributos e as operações dos tipos de objetos, sem se preocupar com a implementação;

- *Modelos de Interações*: representados pelos diagramas de seqüência, baseados nos Modelos de Casos de Uso;

- *Framework de Modelos:* são projetados em um alto nível de abstração, estabelecendo um esquema genérico que pode ser importado, com substituições e extensões, de modo a gerar aplicações específicas [D'Souza, 1999]. Neste modelo são mostrados os tipos que podem ser estendidos, indicando-os com os sinais < > (*placeholders);* e

- *Modelo de Aplicação do Framework:* específico para modelar *frameworks*. Representa a dependência dos tipos do *framework* com os tipos estendidos na aplicação. Neste modelo, os tipos com *placeholders* são substituídos pelos seus respectivos tipos.

No nível do ***Projeto Interno dos Componentes*** dá-se ênfase na implementação dos requisitos especificados para os componentes do sistema, preocupando-se com suas distribuições físicas. Neste nível utiliza-se a técnica:

- *Modelo de Classes:* modelo que representa as classes com seus atributos, operações e relacionamentos. Esse modelo é derivado do *Modelo de Tipos* do nível *Especificação dos Componentes*; e o

- *Modelo de Componentes:* que define a organização e a dependência entre os componentes.

2.4.3 Método *UML Components*

Segundo Cheesman [Cheesman , 2000], o método *UML Components* representa um processo para a especificação de software baseado em componentes. Neste processo, os sistemas são estruturados em quatro camadas distintas:

- *interface com o usuário*,
- *comunicação com o usuário*,
- *serviços de sistemas*, e
- *serviços do negócio*.

O relacionamento entre estas camadas representa a arquitetura do sistema.

O *UML Components* propõe a utilização da UML para modelar todas as fases do desenvolvimento de sistemas baseado em componentes, compreendendo as atividades de definição dos requisitos, identificação e descrição das interfaces entre os componentes, especificação, implementação e montagem dos componentes. Além disso, os autores tiveram a preocupação de oferecer uma solução sem especificar uma particular plataforma de desenvolvimento.

UML Components: ênfase na especificação de componentes para serem reutilizados.

Outros métodos de DBC, não abordados neste trabalho, também vêm sendo alvos de pesquisas e sucessivos refinamentos. Dentre estes métodos, encontra-se o CBD96 [CBD-HQ, 2002], *Rational Unified Process* (RUP) [Jacobson, 2001], *Select Perspective* [Perspective, 2002], o COmponent-based METhodology (COMET) [COMET, 1997] e o KobrA [Kobra, 2000].

Esta seção introduziu os principais métodos de DBC disponíveis na literatura. A seção seguinte descreverá os conceitos sobre ferramentas CASE.

2.5 Ferramentas CASE

O Instituto de Engenharia de Software americano [SEI, 1999a] relata que, desde os primeiros dias das atividades de desenvolvimento de software, já existia uma consciência da necessidade de utilização de ferramentas para auxiliar o engenheiro de software na otimização de suas tarefas. Inicialmente, a concentração estava em ferramentas de suporte à execução dos programas, como tradutores, compiladores, montadores e processadores de macro. Entretanto, com o crescimento dos recursos computacionais aliados à complexidade do software, motivou-se a necessidade de expansão dessas ferramentas com novas funcionalidades. Em particular, a utilização de sistemas interativos de tempo compartilhado no desenvolvimento de software encorajou a criação de editores de programas, *debuggers* e analisadores de código.

À medida que os computadores iam se tornando mais confiáveis e mais utilizados, a necessidade de uma ampla noção do desenvolvimento de software tornava-se aparente. Assim, o desenvolvimento de software passou a ser visto como [SEI, 1999a]**:**

- Uma atividade de larga escala que envolve esforços significativos, a fim de identificar os requisitos, projetar uma solução apropriada, implementar esta solução, testa-lá e, finalmente, documentar as funcionalidades do produto final antes de se colocar em produção;
- Um processo a longo prazo, no qual o seu principal produto, o software, necessita de melhorias contínuas durante todo o seu ciclo de vida. As implicações dessa visão são que a estrutura do software deve permitir a inclusão de novas funcionalidades de forma facilitada e, além disso, as informações referentes aos requisitos, projeto,

implementação e testes, devem ser armazenadas e disponibilizadas de forma detalhada, a fim de auxiliar as tarefas de manutenção. Adicionalmente, as diversas versões dos artefatos produzidos durante todo o projeto devem ser mantidas, de modo que qualquer informação necessária para o grupo de desenvolvimento possa ser obtida sem maiores problemas; e

- Uma atividade coletiva, que envolve a interação entre um grupo de participantes durante cada etapa do ciclo de vida. Assim, esses participantes devem poder cooperar entre si, de maneira controlada, e possuir visões consistentes de determinados aspectos do projeto.

A visão de desenvolvimento de software como um processo de larga escala e em longo prazo resultou no desenvolvimento de uma variedade de ferramentas que ofereciam suporte a esta tarefa [Humphrey, 1989]. Inicialmente, as ferramentas eram limitadas, não oferecendo muitos recursos a este suporte. Entretanto, dois importantes avanços resultaram na melhoria das funcionalidades dessas ferramentas, a saber [SEI, 1999a]:

- Pesquisas na área de processos de desenvolvimento de software, as quais deram origem ao crescimento do número de métodos de projeto (por exemplo, programação e análise estruturada), que passaram a ser utilizados como base para o desenvolvimento. Esses métodos foram idealmente apropriados para serem automatizados por ferramentas de suporte em cada etapa de sua execução, tendo notações gráficas associadas e uma grande variedade de artefatos associados (diagramas, anotações e documentações), necessários para serem armazenados e documentados.
- Os recursos computacionais oferecidos pelas máquinas. Com o avanço destes recursos, começou-se a explorar, efetivamente, a capacidade de armazenamento de dados, o aumento do poder de processamento e os recursos mais sofisticados, oferecidos pelos mecanismos para visualização gráfica, capazes de exibir gráficos, modelos de análise e projeto e uma série de diagramas.

A partir das funcionalidades obtidas com a utilização destas ferramentas, pôde-se obter uma definição mais precisa do termo ferramenta CASE. Segundo Pressman [2001], ferramentas de engenharia de software apoiadas por computador (*Computer-Aided Software Engineering*, CASE) auxiliam gerentes e profissionais de engenharia de software em todas atividades associadas com o processo de desenvolvimento. Automatizam as atividades de gestão de projetos, gerenciam todos os artefatos produzidos ao longo do processo e assistem aos engenheiros de software em seu trabalho de análise, projeto, implementação e teste.

A área de CASE tem se desenvolvido muito rapidamente e, conseqüentemente, diferentes terminologias são usadas para descrever estas ferramentas. Termos como ferramentas, *toolkits*, *workbenches* e ambientes têm sido utilizados de maneira inconsistente. Assim, uma taxonomia de ferramentas CASE torna-se extremamente necessária.

Diversas taxonomias podem ser encontradas na literatura. Em [SEI, 1999a], os autores dividem as ferramentas em três grupos:

- Ferramentas de natureza interativa (por exemplo, uma ferramenta que ofereça suporte a um determinado método de desenvolvimento) e as que não se encaixam nesta categoria (por exemplo, um compilador). As primeiras classes são, algumas vezes, chamadas de ferramentas CASE, enquanto que as últimas são chamadas de ferramentas de desenvolvimento.
- Ferramentas que oferecem suporte às atividades iniciais do ciclo de vida do projeto de software (como análise de requisitos) e as que são utilizadas em fases posteriores do ciclo de vida (como compiladores e ferramentas de teste). As primeiras classes são, algumas vezes, chamadas de ferramentas CASE *front-end*, enquanto que as últimas são chamadas de *back-end*.
- Ferramentas que são utilizadas para uma etapa específica do ciclo de vida de desenvolvimento (como, por exemplo, uma ferramenta de análise de requisitos ou codificação) e as que são comuns a uma série de etapas do ciclo de vida (como, por exemplo, ferramentas de documentação e gerenciamento de configuração).

Sommerville [2000] e Pressman [2001] definem uma taxonomia de ferramentas CASE mais abrangente, utilizando como base o critério de função. A partir deste critério, as ferramentas são agrupadas de acordo com dezesseis funcionalidades, incluindo engenharia de processo de negócio, modelagem e gestão de processo, análise de risco, entre outras. Esse critério é o que será utilizado neste trabalho.

A seção seguinte apresenta a ferramenta MVCASE que se encaixa na taxonomia de análise e projeto, segundo o critério de função e que oferece suporte ao desenvolvimento de software baseado em componentes.

2.5.1 Ferramenta MVCASE

A Ferramenta MVCASE [Lucredio, 2001] é uma CASE que provê técnicas gráficas e textuais para o engenheiro de software. Com efeito, este especifica o sistema, em um alto nível de abstração, e gera código dessas especificações, por exemplo, em linguagens como *Java*, usando os benefícios da UML, a qual permite ao desenvolvedor trabalhar com diferentes visões no desenvolvimento de sistemas.

As especificações criadas são armazenadas em descrições textuais em uma linguagem de modelagem. Estas descrições são as representações dos modelos do sistema, contendo cada parte do mesmo.

A Ferramenta MVCASE é totalmente escrita na linguagem *Java*, usando a tecnologia Java 2, da Sun Microsystems, e a *Java Foundation Classes* (JFC), a qual oferece suporte a interface livre de plataforma.

No estágio atual, a MVCASE suporta técnicas para persistência e criação de objetos distribuídos, em uma arquitetura de três camadas. Usando os três níveis da especificação, o desenvolvedor pode separar os objetos em objetos clientes, objetos contendo regras de negócio e objetos que provêem serviços, como um banco de dados. Estes objetos podem ser distribuídos em diferentes plataformas, criando aplicações cliente e servidor em um alto nível de abstração.

Esta tecnologia permite desenvolver aplicações reutilizando componentes, deixando tarefas como conectividade, segurança e transações gerenciadas pelo servidor, o qual as realiza de forma transparente para o usuário ou o desenvolvedor.

Segundo os autores da UML [Rumbaugh, 1991], os diagramas podem ser divididos em dois grupos: estruturais e comportamentais, conforme mostram, respectivamente, as Tabelas 2 e 3.

Tabela 2 - Diagramas Estruturais

Diagramas Estruturais
Diagrama de Classes
Diagrama de Objetos
Diagrama de Componentes
Diagrama de Implantação

Tabela 3 - Diagramas Comportamentais

Diagramas Comportamentais
Diagrama de Caso de Uso

Diagrama de Seqüência
Diagrama de Colaboração
Diagrama de Estados
Diagrama de Atividades

Os diagramas estruturais são utilizados para visualizar, especificar, construir e documentar os aspectos estáticos de um sistema. Os aspectos estáticos de um sistema podem ser encarados como uma representação de seu esqueleto e estruturas relativamente estáveis [Rumbaugh, 1999].

Os diagramas comportamentais são utilizados para visualizar, especificar, construir e documentar os aspectos dinâmicos de um sistema. Os aspectos dinâmicos de um sistema são considerados como uma representação de suas partes que sofrem alterações [Rumbaugh, 1999].

Dentre os principais diagramas estruturais suportados pela Ferramenta MVCASE, destacam-se: o Diagrama de Classes e o Diagrama de Componentes. O Diagrama de Classes representa um conjunto de classes, interfaces e colaborações e seus relacionamentos. A Figura 8 mostra um exemplo deste diagrama.

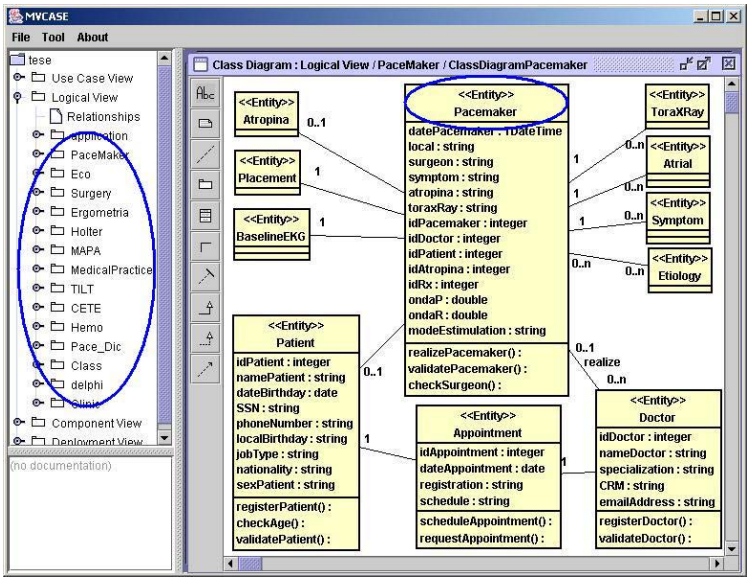


Figura 8 – Diagramas da Classes.

O Diagrama de Componentes mostra um conjunto de componentes, interfaces e seus relacionamentos.

No grupo dos diagramas comportamentais, podem ser citados como primordiais para o desenvolvimento de aplicações, o Diagrama de Caso de Uso e o Diagrama de Seqüência. O Diagrama de Caso de Uso mostra um conjunto de casos de uso e atores e seus relacionamentos, enquanto o Diagrama de Seqüência tem por objetivo mostrar os cenários de execução das operações ao longo do tempo e modelar as conexões de mensagens entre os objetos [Rumbaugh, 1999]. A Figura 9 mostra um exemplo deste diagrama.

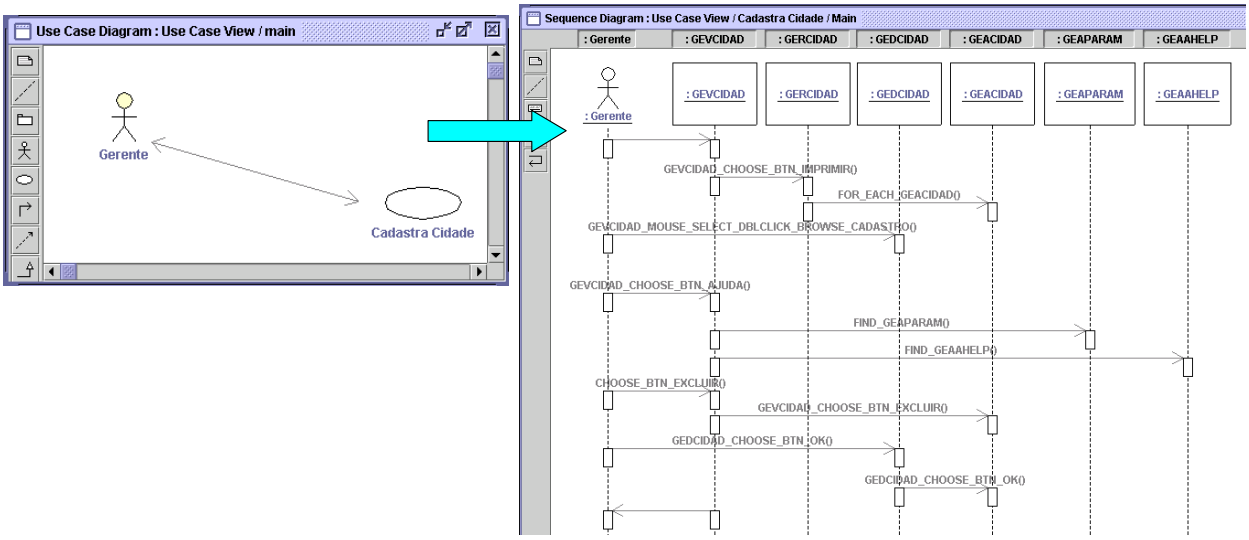


Figura 9 – Diagramas de Caso de Uso e de Seqüência.

Esta seção introduziu as ferramentas CASE consideradas como um elemento primordial no desenvolvimento de software, auxiliando o engenheiro de software em todas as atividades associadas com o processo de desenvolvimento. Também introduziu a taxonomia dessas ferramentas, em especial, a baseada no critério de função como elemento chave. Por fim, foi apresentada a ferramenta MVCASE que suporta a implementação do código *ObjectPascal* utilizada na abordagem para construção e reutilização de componentes, objetivo deste projeto. A próxima seção apresenta Sistemas de Transformação de Software.

2.6 Sistema de Transformação de Software

Sistemas de transformação de software, ou sistemas transformacionais, são ferramentas que permitem a manipulação estrutural e semântica de sistemas. Essas ferramentas têm sido aplicadas a diversos tipos de tarefas tais como síntese de programas e reengenharia, alcançando resultados positivos [NEIGHBORS, 1983], [PRADO, 1992], [LEITE; SANT'ANNA; FREITAS, 1994], [LEITE; SANT'ANNA; PRADO, 1996], [ABRAHÃO; PRADO, 1999], [JESUS; PRADO, 1999], [FUKUDA, 2000], [NOVAIS, 2002], [NOGUEIRA, 2002], [ALVARO et al., 2003].

Sistemas transformacionais podem trabalhar com apenas um conjunto restrito de linguagens de descrição de artefatos e de operadores, ou então, podem ser caracterizados por uma arquitetura que permite ampla configuração para diversas situações de uso. A primeira categoria é chamada de sistemas transformacionais específicos e a segunda de genéricos.

Nesta forma de classificação, compiladores tradicionais são vistos como exemplos de sistemas transformacionais específicos, já que não podem ser configurados para trabalhar com diversas linguagens e também possuem um conjunto pré-configurado e restrito de operadores de transformação.

Os principais esforços realizados em pesquisa e desenvolvimento de sistemas transformacionais concentram-se no aprimoramento de sistemas genéricos.

De maneira abstrata, sistemas transformacionais genéricos podem ser caracterizados por cinco módulos principais: um módulo para a configuração de linguagens de descrição de artefatos, um módulo de descrição de transformadores, um módulo de importação de descrições de artefatos, um Motor Transformacional (MT) e um módulo de exportação de descrições de artefatos. O módulo de configuração de linguagens permite descrever no ambiente diversas linguagens de representação de artefatos, de tal forma que o módulo importador possa capturar estas descrições e submetê-las ao MT. O MT aplica os transformadores que foram disponibilizados através do módulo de descrição de transformadores. O MT pode ter sua lógica de aplicação configurada através de estratégias de uso dos transformadores, de acordo com o grau de automatização suportada pelo ambiente na interação com o Engenheiro de Software que controla a atividade. Uma vez produzida, a descrição resultante é exportada para que o engenheiro de software a utilize. O módulo de exportação realiza essa tarefa, mapeando a forma interna manipulada pelo MT para o formato de apresentação desejado pelo engenheiro de software. O funcionamento dos módulos de importação e exportação está vinculado às linguagens de representação disponibilizadas pelo módulo de descrição de linguagens de representação.

A seguir são descritos alguns dos principais sistemas transformacionais existentes.

- ***Refine*** [REASONING, 1992] é um sistema comercial produzido pela *Reasoning Systems* (US), baseado em pesquisa desenvolvida em *Kestrel*. O principal ponto no *Refine* é uma biblioteca de *Common Lisp* que encapsula as diversas bases de um sistema de transformação. A ênfase atual da *Reasoning* é o uso do *Refine* como uma ferramenta de engenharia reversa e já existem pacotes usando *Refine*, que suporta linguagens tais como: *Cobol*, *Fortran*, *C* e *Ada*;
- ***Popart*** [WILE, 1993] é um sistema com uma linguagem de definição para analisar e reescrever regras. *Lisp* é a linguagem alvo e as regras de rescritas podem ser integradas com ela. A arquitetura do *Popart* e sua integração com *Lisp* fornecem uma poderosa maneira de manipular descrições de software;
- ***Tampr*** [BOYLE, 1989] era também um sistema *Lisp*, mas foi portado para *Fortran*. Ele usa a linguagem *Poly* para a definição de *parsers* e transformações. A maioria das aplicações de *Tampr* tem sido de *Lisp* para *Fortran*, e o código resultante é relatado como muito eficiente. Estes também são os relatos no trabalho com *Pascal* e *C*;
- **TXL** [CORDY J., 1993] é um sistema transformacional desenvolvido no *Software Technology Laboratory do Department of Computing and Information Science* na *Queen's University at Kingston* no Canadá, que com a sua disponibilidade na Internet tem difundido seu uso em diversas aplicações com sucesso;
- **DMS** [BAXTER; PIDGEON, 1997] é um sistema desenvolvido na *Semantic Designs Inc.*, que utiliza a tecnologia de transformação de sistemas para revisar as informações formais do projeto. A estratégia do DMS é descrita na forma de linguagens de domínio e de componentes de software;
- ***RescueWare*** [RESCUEWARE, 2004] é um produto comercial produzido pela *Relativity Technologies*, fundado em fevereiro de 1997, no *Research Triangle Park*, Carolina do Norte, cuja missão é procurar ser o principal fornecedor de soluções para a transformação do software legado. O *RescueWare* se propõe a resolver o problema de compreensão de sistemas legados e mover seletivamente somente as partes relevantes da aplicação para uma plataforma de software mais moderna. Essas plataformas estão baseadas na Internet e na arquitetura *Client/Server*, suportando as linguagens *Java*, *C++* e *Visual Basic*, e
- **JaTS** [CASTOR et al, 2001] é outro sistema transformacional nacional, produzido pela equipe de pesquisadores da Universidade Federal de Pernambuco (UFPE). O *JaTS* visa aumentar a produtividade dos desenvolvedores de software na

implementação de sistemas em Java. A linguagem usada para descrever as transformações é uma linguagem próxima a Java, sendo portanto um superconjunto dela. Outro ponto facilitador da utilização do *JaTS* é que ele leva em consideração semântica da linguagem Java, além de ser uma linguagem genérica. Os trabalhos envolvendo o *JaTS* focam principalmente a implementação de técnicas de *refactoring* [FOWLER, 1999].

Outro importante sistema de transformação que vem sendo utilizado, e que é o mecanismo principal deste projeto é o Draco-PUC, que é apresentado a seguir.

2.6.1 Sistema Transformacional Draco-PUC

O paradigma Draco [NEIGHBORS, 1980] [NEIGHBORS, 1983] está fundamentado no reuso de componentes de software, elementos comuns que estão sempre presentes em diversos sistemas construídos e que refletem objetos e operações inerentes àquela área na qual se está trabalhando.

Por sua vez, estes componentes estão agrupados em domínios, representados por uma linguagem com sintaxe e semântica bem definidas.

A máquina Draco foi construída com o objetivo de testar, desenvolver e colocar em prática o paradigma Draco. Trata-se de um sistema transformacional que se baseia nas idéias de construção de software por transformação orientada a domínios. Um primeiro protótipo da máquina Draco foi construído por Neighbors [NEIGHBORS, 1989]. Posteriormente, a máquina Draco foi reconstruída na PUC-RJ [PRADO, 1992], denominada Draco-PUC, usando linguagens de plataformas de hardware e software consideradas modernas na época.

O Draco-PUC é um sistema transformacional genérico que contém a grande maioria das características importantes em um sistema transformacional. Dentre essas características, destacam-se:

- Um Sistema gerador de analisadores léxicos e sintáticos (*parser*) baseado na sintaxe *Lex/Yacc* [LEITE; SANT'ANNA; FREITAS, 1994] utilizando mecanismo de análise LALR (*Look-Ahead Left Right*) com *backtracking* [FREITAS; LEITE; SANT'ANNA, 1996];
- Um Núcleo transformacional totalmente aberto, que suporta o uso de pontos de controle para o disparo de eventos e direção de fluxo de controle. Mecanismos de controle garantem a exatidão da execução das transformações;

- Uma Linguagem para descrição das transformações, a qual usa transformações locais e globais;
- Mecanismo de casamento de padrões altamente expressivo, fornecendo ao usuário um grande potencial para as especificações dos requisitos de software, utilizando a sintaxe de qualquer linguagem que tenha sido definida no subsistema de *parsers*;
- Facilidade para agrupar transformações em conjuntos com diferentes pontos de controle, e;
- Mecanismo para aplicação dos componentes transformacionais que mapeiam a sintaxe e semântica de uma especificação de entrada para uma nova especificação do mesmo ou de outro domínio. Por exemplo, para geração automática de código em linguagem de programação como *C++*, *Pascal* e *Java*, a partir de um código legado em *Clipper*,por exemplo, ou de uma especificação em uma linguagem de modelagem.

Além disso, o Draco-PUC possui um código portável e que pode operar em diferentes versões de sistemas operacionais, como *Unix*, *Windows 9x*, *NT*, *2000* e *XP*.

Pela estratégia proposta por Prado [PRADO, 1992], é possível a reconstrução de um software pelo "porte" direto do código fonte para linguagens de novos domínios. Para efetuar esse "porte", é necessário construir os domínios das linguagens de origem e alvo do sistema. As regras de transformação, descritas em um transformador, são responsáveis pela automatização ou semi-automatização do processo de construção de software na máquina Draco-PUC.

Um domínio no Draco é definido através de uma Linguagem; esta por sua vez é formada por uma Gramática, um *parser* e *prettyprinter*, ou *unparser*, definidos a partir desta gramática. A figura 10 mostra as partes de um domínio no Draco-PUC.

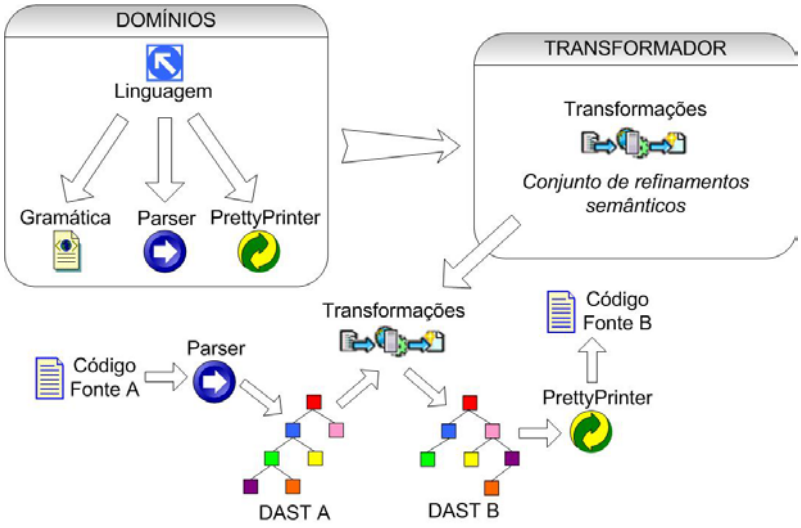


Figura 10: Partes de um domínio do Draco-PUC

Uma Linguagem pode ser definida como qualquer conjunto de sentenças geradas a partir de um conjunto finito de símbolos. É formada por:

- **Gramática** consiste em terminais, não-terminais, um símbolo de partida e produções. Os terminais, ou *tokens* são símbolos básicos a partir dos quais as cadeias são formadas. Os não-terminais são variáveis sintáticas que denotam cadeias de caracteres. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática. Um símbolo de partida, numa gramática, é um não-terminal, e o conjunto que ele denota é a linguagem definida pela gramática. As produções de uma gramática especificam a forma pela qual os terminais e não-terminais são combinados para formar cadeias;
- ***Parser*** gerado, automaticamente, a partir das definições da gramática e do Analisador Léxico da linguagem do domínio, pelo componente *PARGEN* do Draco-PUC. O *parser* é responsável por analisar um programa qualquer deste domínio e gerar, automaticamente, sua representação interna no Draco-PUC. Nesta representação interna, que no Draco-PUC é denominada *Draco Abstract Syntax Tree* (DAST), são aplicadas as transformações, que geram uma nova DAST no mesmo ou em outro domínio;
- ***Prettyprinter*** ou *unparser* é responsável por escrever representações de uma DAST, tornando-a novamente textual na linguagem do domínio. Baseado nas definições das gramáticas o subsistema *PPGEN* do Draco-PUC gera, automaticamente, os respectivos *prettyprinter* dos domínios, e

- **Transformadores**: têm a função de mapear componentes de software. De acordo com o tipo de mapeamento, eles podem ser classificados como:
    - *Transformadores Inter-domínio*: que mapeiam as especificações descritas em uma linguagem de um domínio para descrições de linguagem de outro domínio;
    - *Transformadores Intra-domínio*: que mapeiam estruturas de uma linguagem para estruturas da mesma linguagem do domínio. Tem como objetivo otimizar ou preparar o código para as transformações inter-domínio e ainda, possibilitar múltiplas visões de uma descrição em um mesmo domínio.

Um transformador possui um conjunto de transformações. A escrita das transformações é feita pela definição do padrão de reconhecimento *Left Hand Side* (LHS), e do padrão de substituição *Right Hand Side* (RHS). O LHS define a sintaxe da linguagem de origem para a transformação e o RHS define a sintaxe da linguagem alvo para a transformação. Além dos padrões de reconhecimento e de substituição, uma transformação pode conter outros pontos de controle, aos quais, pode-se associar código para o desempenho de tarefas, relacionadas com pré e pós-condições da transformação.

As transformações no Draco-PUC podem ser escritas com base na própria sintaxe das linguagens sendo transformadas e a linguagem de transformação é orientada a conjunto, o que aumenta o grau de organização das transformações.

Sabe-se então que o primeiro passo para a criação de um domínio é a criação da gramática. A seguir são especificadas as informações necessárias para a descrição de uma gramática no Draco-PUC.

2.6.1.1 Gramática no Draco-PUC

Uma gramática livre de contexto é um sistema formal que descreve uma linguagem especificando como um texto válido pode ser derivado de um símbolo chamado axioma. Ela consiste em um conjunto de regras, do qual em cada estado um determinado símbolo pode ser substituído por uma sucessão de símbolos. Para derivar um texto válido, a gramática é usada como entrada de dados no seguinte algoritmo:

1. Vamos assumir que texto seja uma única ocorrência do axioma.
2. Se não existir nenhuma regra de produção no estado que texto possa ser substituído por uma seqüência de símbolos, então pare.

3. Reescreva texto substituindo um de seus símbolos com uma sucessão de outros símbolos de acordo com alguma regra.
4. Vá para o passo (2).

Quando este algoritmo chega ao fim, o texto é um texto válido na linguagem. A estrutura de frase daquele texto é a hierarquia de sucessões usada em sua derivação. Dado uma gramática livre de contexto que satisfaz certas condições, o Draco-PUC pode gerar um *parser* que executa uma verificação léxica e sintática de programas. O resultado é o código na representação interna do Draco-PUC, a DAST.

2.6.1.2 Estrutura da Gramática no Draco-PUC

A 11 mostra a estrutura de uma gramática no Draco-PUC. Na primeira seção, *"Definições de Precedência"*, o primeiro símbolo *"%"* indica um espaço reservado para a definição das regras de precedência dos operadores. O primeiro símbolo *"%%"* indica o início das regras de produção, sintáticas e léxicas, e o segundo o fim da regras.

```
% Definições de Precedência

%% Início das Regras
Seção de Regras Sintáticas
Seção de Regras Léxicas
%% Fim
```

Figura 11: Estrutura da Gramática no Draco-PUC

A seção de regras sintáticas descreve a gramática livre de contexto como um conjunto de regras de produção. Cada regra de produção consiste em um único nome do lado esquerdo do operador *":"*, uma lista de símbolos do lado direito e um *";"* indicando o fim da regra.

```
(1) statement : variable '=' expression
              ;
(2) statementList :
                  ;
(3) statement : 'if' expression 'then' statement
                'else' statement
              ;
```

A primeira regra de produção afirma que o símbolo *statement* **(1)** pode ser substituído pela sucessão que consiste em três símbolos: *variable*, *'='*, e *expression*. Qualquer ocorrência do símbolo *statementList* **(2)** pode ser substituída por uma sucessão vazia de acordo com a segunda regra de produção. Na terceira regra **(3)**, nota-se que novas linhas podem ser usadas como separadores na descrição de uma regra de produção. Esta notação é chamada de *Backus Naur Form*, ou só BNF.

Símbolos que podem ser substituídos são chamados de não-terminais, e sempre são representados por identificadores (um identificador é uma seqüência de letras e dígitos, onde o primeiro caractere deve ser uma letra). Todo não-terminal deve aparecer antes do operador *":"* em pelo menos uma regra.

Símbolos que não podem ser substituídos são chamados de terminais, e podem ser representados por identificadores ou símbolos literais (um símbolo literal é uma seqüência de caracteres unidos por apóstofres (’). Uma apóstrofe que aparece dentro de um símbolo literal é representada através de duas apóstrofes sucessivas). Um símbolo terminal não pode aparecer antes do *":"* (no lado esquerdo) em nenhuma regra. Terminais representam *strings* de caracteres que são reconhecidas por analisadores léxicos.

Quando uma gramática contém muitas regras que especificam uma substituição do mesmo não-terminal, uma barra vertical (|), que denota alternação, pode ser usada para evitar que o símbolo que está sendo substituído seja re-escrito. A 12 mostra uma alternação que especifica duas regras. O não-terminal a ser substituído é o *statement* em cada caso. Possíveis sucessões de substituição estão separadas através de barras verticais.

```
%%
statement       : variable '='
expression
                | expression
                ;

expression      : expression '+'
expression
                | expression '-'
expression
                | expression '*'
expression
                | expression '/'
expression
                ;
```

Figura 12: Gramática de uma calculadora com expressões

Como definir regras recursivas

Para analisar gramaticalmente uma lista de itens de tamanho indefinido, pode ser escrita uma regra recursiva, que está definida sobre si mesma. Por exemplo, esta regra analisa uma possível lista vazia de *func_params*:

```
functions : iden '(' func_params? ')'
          ;
```

Para definir uma regra que analise uma lista de zero ou mais elementos usa-se:

```
frame_decl : declarations*
           ;
```

E, para uma lista de um ou mais elementos usa-se:

```
frame_decl : declarations+
           ;
```

Para definir uma regra que é uma lista de um ou mais elementos intercalados por uma vírgula (,):

```
decl : button++','
     ;
```

Regras Léxicas

As produções da seção de regras léxicas são compostas por expressões regulares. Uma expressão regular é um padrão de descrição usando uma metalinguagem, usada pra descrever um padrão específico do seu interesse. Os caracteres que formam uma expressão regular são apresentados na Tabela 4.

Tabela 4: Caracteres de uma expressão regular

Caracter	Descrição
.	Reconhece qualquer caractere exceto o caractere de nova linha (“\n”)
*	Reconhece zero ou mais cópias da expressão anterior
^	Reconhece o início de uma linha como o primeiro caractere de uma expressão regular. É também usado como negação dentro dos colchetes.
[ ]	Uma classe de caracteres o qual reconhece qualquer caractere dentro dos colchetes. Se o primeiro caractere for um (“^”) ele troca o sentido para reconhecer qualquer caractere, exceto os dentro dos colchetes.
&	Reconhece o fim de uma linha como o último caractere de uma expressão regular.
{ }	Indica quantas vezes o modelo anterior pode reconhecer quando contém um ou mais números. Ex: A(1,3)
\	Usado para imprimir meta-caracteres, como partes de uma usual seqüência de caracteres especiais. Ex: “\n” ou “*”
+	Reconhece uma ou mais ocorrências da expressão regular anterior. Ex: [0-9]+
?	Reconhece uma ou mais ocorrências da expressão regular anterior. Ex: -?[0-9]+
|	Reconhece uma ou mais ocorrências da expressão regular seguinte. Ex: ab|cd|ef
...	Interpreta tudo que estiver dentro da aspas (meta-caracteres diferentes do C perdem significado)
/	Une a expressão regular precedentes mas somente se seguido por uma expressão regular. Ex: 0/1
( )	Grupos de uma série de expressões regulares juntos dentro de uma nova expressão regular. Parênteses são úteis quando construídos com alto padrão de complexidade, ou seja, com , + e |. Ex: $(0^{+}\|1^{*})$

No Draco-PUC, as regras léxicas definidas por *IGNORE* são ignoradas no *parsing*. A Figura 13 também mostra este tipo de regra para ignorar espaços em branco e comentários.

```
NUMERO          : [0-9]+(\.[0-9]+)?(eE[+\-]?[0-9]+)?
                ;
IDENTIFICADOR   : [A-Za-z_][A-Za-z_0-9]*
                ;
STRI            : \'([^\']|\'\')+\'
                ;
STRI            : \"([^\"]|\"\")+\"
                ;
IGNORE          : [\t]+          /* ignora espaços em branco */
                ;
IGNORE          : "//"[^\n]+     /* ignora comentários */
                ;
IGNORE          : "&&"[^\n]+    /* ignora comentários */
                ;
```

Figura 13: Seção de regras léxicas

A regra *NUMERO* **(1)** especifica um ou mais dígitos, *[0-9]+*, seguidos de uma parte de fração opcional, *(\ .[0-9]+)?*, que consiste de um ponto decimal e pelo menos um ou mais dígitos, opcionalmente seguido de uma parte exponencial, *([eE][+ \ -]?[0-9]+)?*, que consiste ou de um *“e”* ou um *“E”*, um sinal de adição ou subtração opcional e pelo menos um ou mais dígitos. Após a definição de cada regra é necessário adicionar dois *TABS*.

Precedência, Associatividade e Declaração de Operadores

Precedência controla quais operadores executar primeiro em uma expressão. Tradições matemáticas dizem que a multiplicação e divisão possuem precedência mais alta do que a adição e subtração, assim *a+b*c* significa *a+(b*c)*. Em qualquer gramática de expressões matemáticas, os operadores são agrupados em níveis de precedência de mais baixo para mais alto. O número total de níveis depende da linguagem.

Associatividade controla o agrupamento de operadores no mesmo nível de precedência. Operadores podem se agrupar à esquerda, por exemplo, *a-b-c* em *C* significa *(a-b)-c*, ou à direita, por exemplo, *a=b=c* em *C* significa *a=(b=c)*.

No Draco-PUC a precedência e associatividade dos operadores é declarada na seção de *Definições de Precedência*, como mostra a Figura 14. As declarações possíveis são *"% left"* e *"% right"* que tornam um operador associativo à esquerda ou à direita, respectivamente.

```
% left '+' '-'
% left '*' '/'
% right '='
%%
sentença              : identificador '=' expressao
                      | expressao '*' expressao
                      | expressao '-' expressao
                      | expressao '/' expressao
                      | expressao
                      ;
IDENTIFICADOR         :[0-9]+(\.[0-9]+)?(eE[+\-]?[0-9]+)?
                      ;
IGNORE                : [\t]+
                      ;
%%
```

Figura 14 Precedência de Operadores

Operadores são declarados em ordem crescente de precedência. Todos os operadores declarados na mesma linha são do mesmo nível de precedência. Por exemplo, na Figura 14 os operadores de mais baixa precedência são *"+"* e *"-"*, os de precedência média são o *"*"* e *"/"*, e o de mais alta precedência é o *"="*.

Adicionando regras de formatação de *layout* na gramática

O Draco-PUC gera automaticamente o *prettyprinter* de uma linguagem a partir da sintaxe da linguagem do domínio. Para cada regra existente na gramática, são criadas regras

de exibição de DASTs. Para melhorar a legibilidade dos programas, regras de formatação de *layout* foram adicionadas ao Draco-PUC. Estas regras são descritas pelas primitivas indicadas na Tabela 5.

Tabela 5: Primitivas do *prettyprinter* usadas na formatação

Primitiva	Descrição
"*str*"	Imprime uma *string* (*str*) literal
.NL	Gera uma nova linha
.COL(*n***)**	Posiciona o cursor na coluna *n*. Se a coluna corrente é maior do que *n*, uma nova linha será criada.
.LM	Fixa margem à esquerda na posição corrente.
.LM (- <*number*>**)**	Fixa a margem à esquerda a partir da margem original menos o argumento.
.LM (+ <*number*>**)**	Fixa a margem à esquerda a partir da margem original mais o argumento (o sinal de + é opcional).
.SLM	Posiciona o cursor na coluna definida como margem à esquerda. Se a coluna é maior do que a margem, uma nova linha será criada.
.SLM (<*number*>**)**	Faça *.SLM* se a coluna impressa for maior do que a dada coluna.
#*N*	Imprime o *n-ésimo* filho do nó.
.TREEPRINT (*n, sep, trm***)**	Exibe um nó com um número variável de filhos. A exibição começa no *n-ésimo* filho. O parâmetro *sep* define uma série de primitivas que serão executadas entre a exibição de cada filho. O último parâmetro indica uma seqüência a ser executada após a impressão de cada filho.

A Figura 15 apresenta um trecho da gramática e do *prettyprinter* da linguagem Java gerados **sem** as regras de formatação de *layout*.

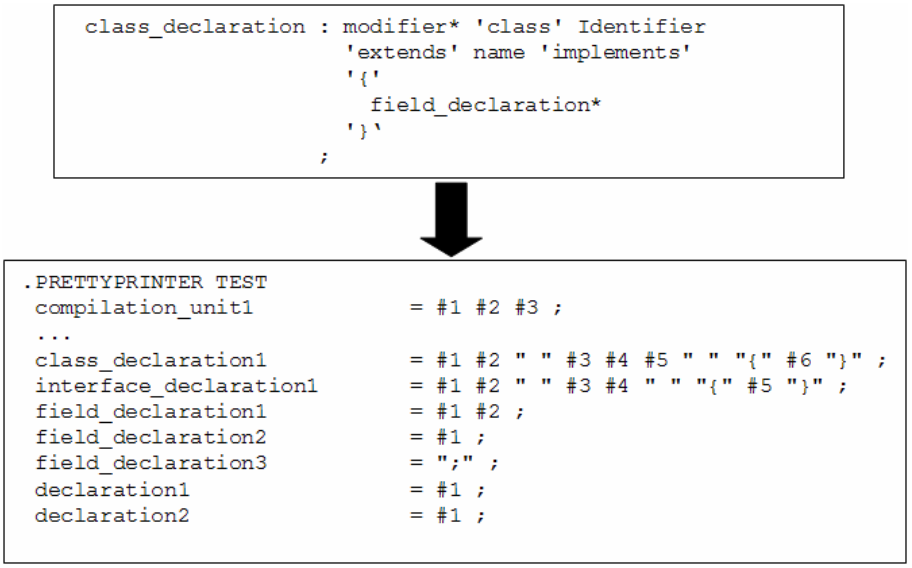


Figura 15: *Prettyprinter* Java sem as regras de formatação de *layout*

Nota-se, na Figura 15, que há uma linha para cada palavra-chave na forma interna. Os *#1* nas linhas anteriores recorrem à primeira entrada na forma interna do nó após a palavra-chave. As *strings* entre apóstrofes serão impressas literalmente.

A Figura 16 apresenta o mesmo trecho de gramática e *prettyprinter* da Figura 15 porém, **com** as regras de formatação de layout.

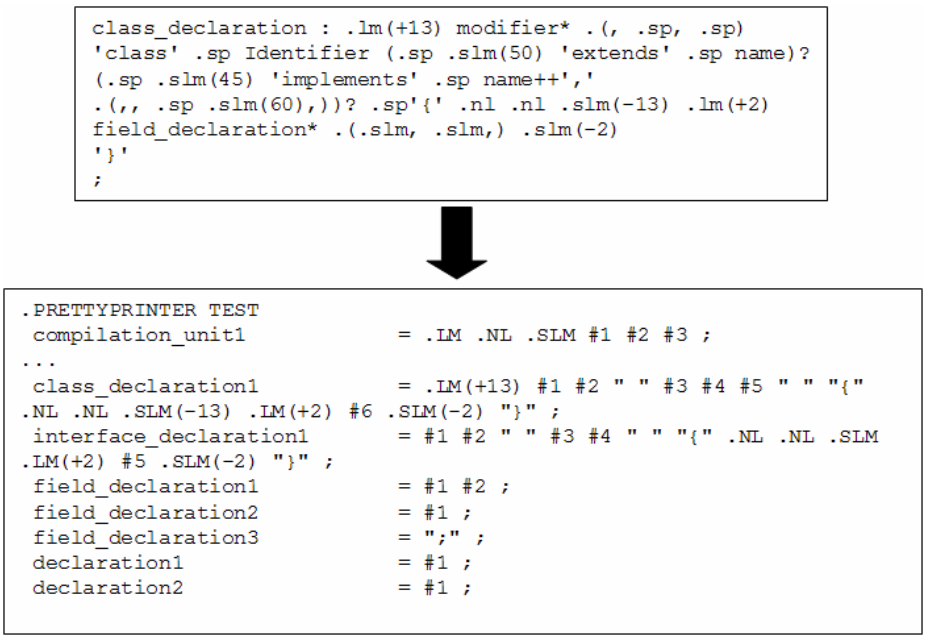


Figura 16: *Prettyprinter* Java com as regras de formatação de *layout*

Os parênteses na Figura 16 são usados para descrever diretivas que atuam em elementos de uma lista. As seqüências de diretivas são separadas por vírgulas e só têm efeito se possuírem mais de um elemento.

O *.LM(+13)* na regra *class_declaration1* determina a margem esquerda como sendo a da posição corrente mais 13.

O *.SLM(50)* executará o *.SLM* somente se a posição da coluna corrente for maior do que o argumento.

2.6.1.3 Transformadores no Draco-PUC

Os transformadores são pacotes de transformações que atuam numa DAST, para gerar uma nova DAST. As transformações são escritas com comandos da meta-linguagem do subsistema *TFMGEN* [Prado, 1992] do Draco-PUC. Esta meta-linguagem possui comandos que atendem a quase todas as necessidades de um sistema de transformação. Usando

comandos do *TFMGEN* é possível descrever as transformações orientadas pelas sintaxes e semânticas das linguagens de origem e alvo das transformações. Existem dois tipos de transformações, definidas nos transformadores, são elas:

- **Intra-Domínio**: correspondem as transformações horizontais, e mapeiam estruturas de uma linguagem para estruturas na mesma linguagem do domínio. Normalmente são utilizadas para otimização ou organização de programas; e
- **Inter-Domínios**: correspondem as transformações verticais, e mapeiam as aplicações descritas em uma linguagem de um domínio para descrições de linguagem de outro domínio. Essas promovem a mudança de domínio.

As transformações são responsáveis pela automatização ou semi-automatização do processo de construção de software, através da otimização e manipulação da DAST. Uma transformação deve ser composta basicamente de um padrão de reconhecimento, chamado LHS (*Left-Hand-Side*) e um padrão de substituição, chamado de RHS (*Right-Hand-Side*).

A Figura 17 a seguir, apresenta o funcionamento e a diferença entre as transformações intra-domínios e as inter-domínios a partir de uma transformação de um programa na linguagem A para a linguagem B.

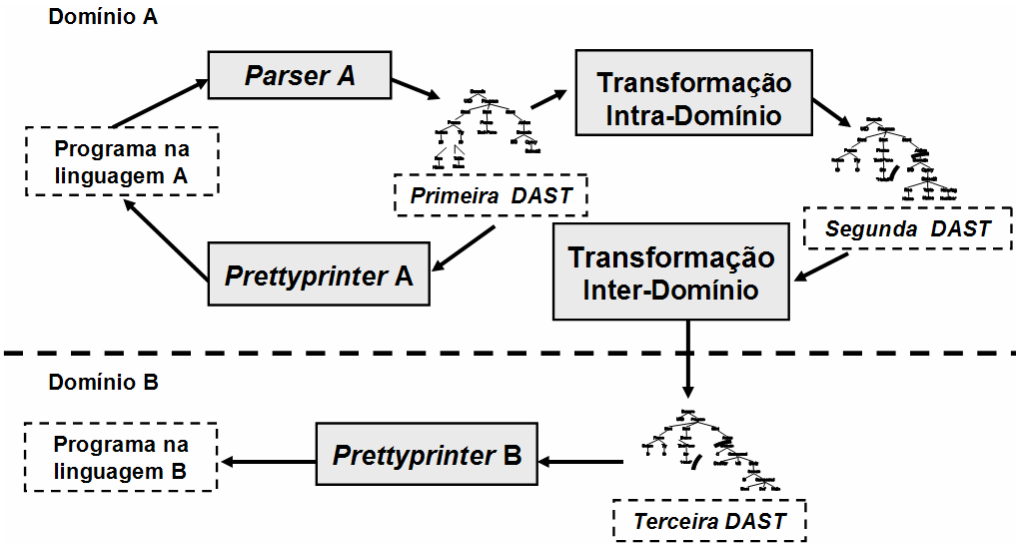


Figura 17: Transformações Intra-Domínios x Inter-Domínios

Ao submeter o *programa na linguagem A* ao *parser*, uma *Primeira DAST* é gerada. A partir dessa DAST, é possível obter novamente o *programa na linguagem A* através do *prettyprinter A*. A *Transformação Intra-Domínio* atua nesta primeira DAST gerada para, a

partir de seus mapeamentos semânticos, gerar uma *Segunda DAST*. Uma outra transformação, esta *Inter-Domínios*, vai atuar nesta segunda DAST para obter uma *Terceira DAST*. A partir desta última DAST, o *Prettyprinter da linguagem B* obtém o *Programa na linguagem B*.

O Draco-PUC suporta o uso de *pré-condições* e *pós-condições* em suas transformações, através de pontos de controle genéricos sobre regras, conjuntos e transformadores. Cada ponto de decisão possui um mecanismo de escape que permite a comunicação com elementos externos. A comunicação é feita através de comandos descritos na linguagem *C++*. A organização destes pontos de controle é representada no *framework* da Figura 18, onde cada um dos elementos de uma solução transformacional em Draco-PUC está representado, juntamente com seus pontos de controle.

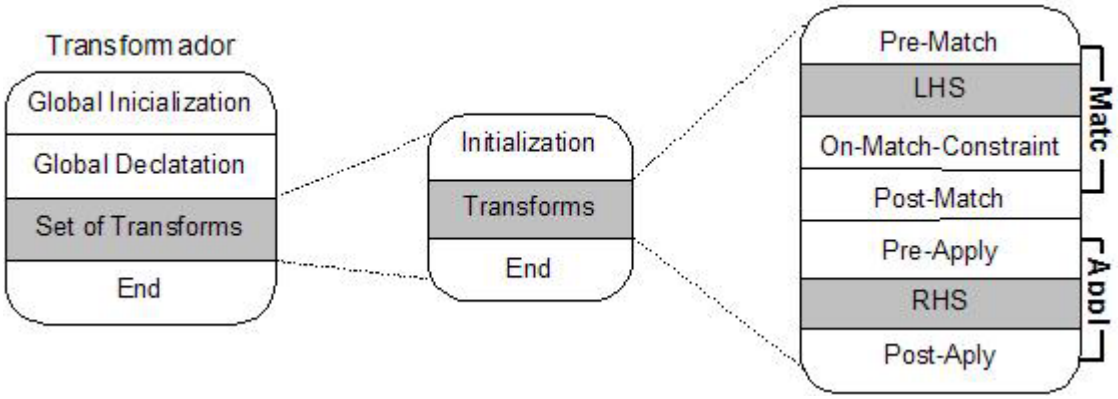


Figura 18: *Framework* para transformações do Draco-PUC

Para a fase de tentativa de aplicação de um padrão de busca (fase de *match* de um *transform*), estão disponíveis os pontos de controle *pre-match*, *match-constraint* e *post-match*. Para a fase de aplicação de um padrão de substituição (fase de *apply* de um *transform*), estão disponíveis os pontos de controle *pre-apply* e *post-apply*. Para um conjunto de transformações (*Set Of Transforms*), estão disponíveis os pontos de controle *initialization* e *end*. Para um transformador (*transformer*) estão disponíveis os pontos de controle *declaration*, *initialization* e *end*.

Um transformador contém as seguintes seções:

- *Global-Initialization*: área reservada para as inicializações globais. É executado quando o transformador for selecionado para a aplicação;
- *Global-Declaration*: área reservada para declaração de variáveis globais; e
- *End*: executado quando o transformador terminar de ser aplicado.

O *Set of Transforms*, é um conjunto de transformações, que possuem os seguintes pontos de controle:

- *Initialization*: executado sempre que o conjunto de transformações for selecionado para aplicações; e
- *End*: executado sempre que o conjunto de transformações terminar de ser aplicado.

- Cada *Transform* pode conter os seguintes pontos de controle:
- *Pre-Match*: executado toda vez em que a regra de transformação é testada sobre um trecho de descrição de entrada;
- *On-Match-Constraint*: executado logo após ter sido finalizada uma tentativa de casamento entre o lado esquerdo (LHS) e um trecho da descrição de entrada;
- *Post-Match*: executado após a regra de transformação ser testada sobre um trecho da descrição de entrada;
- *Pre-Apply*: executado imediatamente antes da substituição do trecho de entrada selecionado pelo lado direito (RHS) e
- *Post-Apply*: executado após a substituição do trecho de entrada selecionado pelo lado direito (RHS).

Os conjuntos de transformações também permitem especificar como as suas transformações serão aplicadas na DAST do programa alvo das transformações, de cima para baixo (*Top-Down*), de baixo para cima (*Bottom-Up*) ou seqüencialmente (*Sequential*). A aplicação das transformações pode ser em um único passo (*Single Step*), para transformações Inter-Domínios, ou em vários passos (*Exhaustive*) para transformações Intra-Domínio.

Eventualmente, uma transformação necessita de informações capturadas por outras transformações. Se essa informação for composta de muitas linhas de código, áreas de trabalho chamadas de *Workspaces* podem ser usadas. A estratégia de transformação por *Workspaces*, por exemplo, utiliza os pontos de controle para adiar a reescrita de uma regra para fazer uma difusão ou concentração de uma descrição de uma linguagem para outra. Assim, uma transformação, desprovida de LHS, pode passar meta variáveis através de um *Template*, que é um padrão formatado de uma dada categoria sintática, armazenando-o em um *Workspace*. O *Template* armazenado pode então ser reutilizado por outras transformações.

Caso necessário, nos pontos de controle, o Engenheiro de Software pode adicionar ações semânticas, escritas na linguagem *C++*. Dessa forma, além dos *Workspaces*, pode-se contar com todos os recursos da linguagem *C++*, que permite resolver problemas específicos não previstos no *Framework* das transformações.

Para facilitar o tratamento de informações semânticas, o sistema transformacional Draco-PUC dispõe de um recurso de Base de Conhecimento (KB - *Knowledge Base*), que captura informações sobre o programa do domínio analisado, e libera o Engenheiro de Software, em alguns casos, do uso de estruturas de dados auxiliares. Esse recurso armazena cláusulas, fatos e regras, em uma notação, similar a *Prolog*, que serão consultadas posteriormente. Comandos utilizados nos pontos de controle dos componentes permitem armazenar, consultar, recuperar e excluir fatos ou regras na KB, como por exemplo, *KBAssert()*, *KBAssertIfNew()*, *KBSolve()*, *KBRetrieve()* e *KBDelete()*, como mostra a Tabela 6.

Tabela 6: Pontos de Controle da KB

Pontos de Controle	**Descrição**
KBAssert (X)	Insere o fato *(X)* na KB
KBAssertIfNew (X)	Insere o fato *(X)* na KB se ainda não existir
KBDelete (X)	Apaga o fato *(X)* da KB
KBSolve (X)	Realiza uma consulta, onde *(X)* é uma *string* que contém a consulta. *KBSolve* retorna o valor lógico *Verdadeiro* caso a consulta seja bem sucedida
KBRetrieve (X)	Acessa um elemento da consulta *(X)*
KBWrite (X)	Grava fisicamente o valor *(X)* na KB
KBRead (X)	Lê a KB

A construção das transformações dispõe de outros recursos, que permitem a comunicação de funções e macros do Draco-PUC com a linguagem *C++* e vice-versa, uma vez que o transformador é construído em *C++*. Um exemplo é a função *expand* que exporta uma metavariável do Draco-PUC como *string* para ser usada como uma variável *C++* num ponto de controle, e a função *Set_Leaf_Value* que permite associar uma metavariável do Draco-PUC a uma *String* com valor pré-definido armazenado em uma variável *C++*.

Uma transformação pode também, por meio do comando *Apply*, chamar outra transformação ou um conjunto de transformações passando metavariáveis como parâmetro. Esses mecanismos são graficamente apresentados na Figura 19 (SANT'ANNA, 1999b), onde as setas indicam a dependência entre as transformações ou conjuntos de transformações. Assim, por exemplo, o "*Transform 1"* pode passar informações para o "*Template 1"*, as variáveis globais, a "*Knowledge Base"*, o "*Transform 2"* e outros "*Set of Transforms"*. Já o

"*Transform 2"* usa as informações do "*Template 1"*, das variáveis globais e diretamente do "*Transform 1"*; e o "*Template 2"*, usa informações do "*Set of Transforms"*.

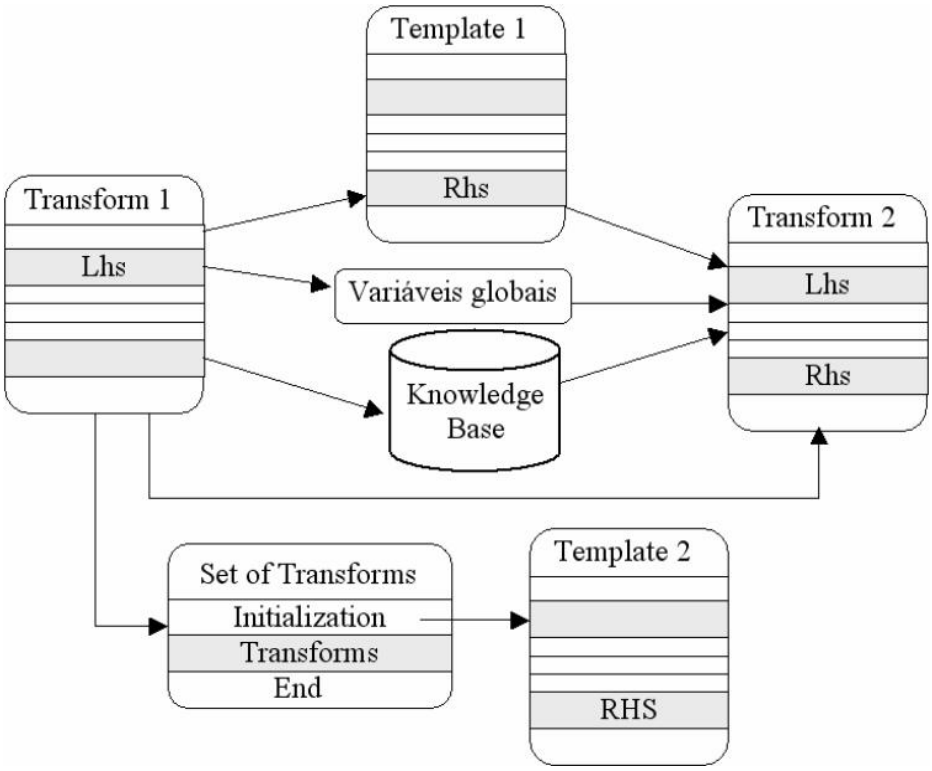


Figura 19: Comunicação entre Transformadores

Todos esses recursos são importantes para suportar a transformação de software orientada a domínios e necessita de mecanismos para executar o refinamento automático de descrições em diferentes níveis de abstração [SANT'ANNA, 1999a].

Procurando generalizar os transformadores de software, segundo a idéia de componentes de software, deixa-se de lado a visão clássica dirigida por conjuntos de regras de reescrita. Pode-se pensar num transformador como uma composição de elementos heterogêneos que recebe em sua entrada uma descrição de um artefato de software e produz em sua saída uma descrição transformada. Nesta visão, regras de reescrita são apenas um dos tipos de componentes. O modelo típico de sistemas transformacionais passa a ser uma instância deste modelo, no qual existe um processador único, o motor transformacional, centralizando o controle de aplicação de regras de transformação que fazem parte de um transformador [SANT'ANNA, 1993].

O Sistema de Transformação (ST) Draco-PUC é usado como principal mecanismo para a geração de código dos componentes e de suas aplicações. Para suportar a geração do código são construídos dois domínios denominados *MDL* (*Modeling Domain Language*) [Fukuda,2000] e *ObjectPascal* [Moraes 2001].

As especificações na linguagem de modelagem *MDL*, modeladas na ferramenta CASE, são armazenadas num arquivo de descrições textuais. Baseado na linguagem destas descrições textuais foi construído o domínio de modelagem *MDL*.

Para gerar o código *ObjectPascal* a partir de descrições *MDL*, foi também construído o domínio *ObjectPascal* no ST Draco. As transformações que geram código baseiam-se nas gramáticas dos dois domínios, *MDL* e *ObjectPascal*, apresentadas parcialmente na Figura 20.

Para que se tenha uma idéia do mapeamento no ST Draco, na Figura 20 tem-se, à esquerda, as regras de produção que identificam uma classe (1) com seus métodos (2) e atributos (3) na linguagem *MDL* e, à direita, as correspondentes regras na linguagem *ObjectPascal*.

MDL	***ObjectPascal***
class_Object : '(object' .sp 'Class'.sp STRI ')'; **(1)** **operations :** '(object' .sp 'Operation' .sp STRI ')'; **(2)** **classAttribute :** '(object' .sp 'ClassAttribute' .sp STRI ')'; **(3)** **STRI :** \"([^\"]|(\\[^\n]))*\"; . . .	**unit_heading :** *'UNIT'* IDENTIFIER *';' ;* **(1)** **procedural_type_decl :** 'PROCEDURE' .sp IDENTIFIER .nl; **(2)** **not_property_definition :** 'PROPERTY' .sp IDENTIFIER ';'; **(3)** **IDENTIFIER :** *[A-Za-z_][A-Za-z_0-9]*;* . . .

Figura 20. Gramáticas *MDL* e *ObjectPascal*.

Na Abordagem, além destes mecanismos, utilizou-se também uma ferramenta RAD (*Rapid Application Development*), conhecida no mercado como *Delphi*, para auxiliar o engenheiro de software no desenvolvimento das aplicações utilizando componentes, apresentada a seguir.

2.7 Linguagem ObjectPascal (Delphi)

Delphi é um ambiente integrado para desenvolvimento de aplicações (IDE - *Integrated Development Environment*) que fornece ferramentas para desenhar, codificar, compilar, executar, depurar, distribuir e testar aplicações, nas plataformas Windows e Linux (Kylix). Dentre suas principais características destacam-se:

- **Desenvolvimento Visual** - Permite a construção visual da interface da aplicação através de uma biblioteca de classes (API - *Application Program Interfac*e) denominada de VCL *(Visual Component Library)*, a qual representa os componentes de interface do Windows (ex. janelas, menus, botões, caixa de edição, e outros).
- **Orientação a Objetos** - Utiliza uma linguagem de programação compilada e fortemente tipificada, denominada *ObjectPascal*. Esta linguagem é uma evolução do Pascal padrão, com características do paradigma da orientação a objetos.
- **Alta produtividade** - Fornece modelos *(templat*es) e *wizard*s para auxiliar na criação de partes da aplicação ou componentes.
- **Edição Two-Way** - Permite a edição visual e sincronizada dos componentes. Quando um componente é editado visualmente, o código fonte é automaticamente atualizado para corresponder à edição visual. E quando o código fonte é editado, a edição visual do componente é automaticamente atualizada.

A VCL *(Visual Component Library)* é uma hierarquia de classes, escritas em *ObjectPascal*, que suporta a criação de componentes, através do mecanismo de herança. Na arquitetura VCL existem dois tipos de componentes que podem ser criados: Visuais, com visibilidade em tempo de projeto (*designtime)* e em tempo de execução (*runtime)*, e Não-Visuais, visíveis apenas em tempo de projeto. A abordagem proposta utiliza a hierarquia de classes da VCL tanto para construção como para reutilização dos componentes.

A Figura 21 mostra a hierarquia base da estrutura de classes da VCL para construção de novos componentes. Todos os componentes descendem da classe TComponent. A classe TComponent herda da classe TPersistent, que por sua vez herda da classe TObject, que é a classe raiz da VCL e de todos os objetos criados no ambiente *Delphi*.

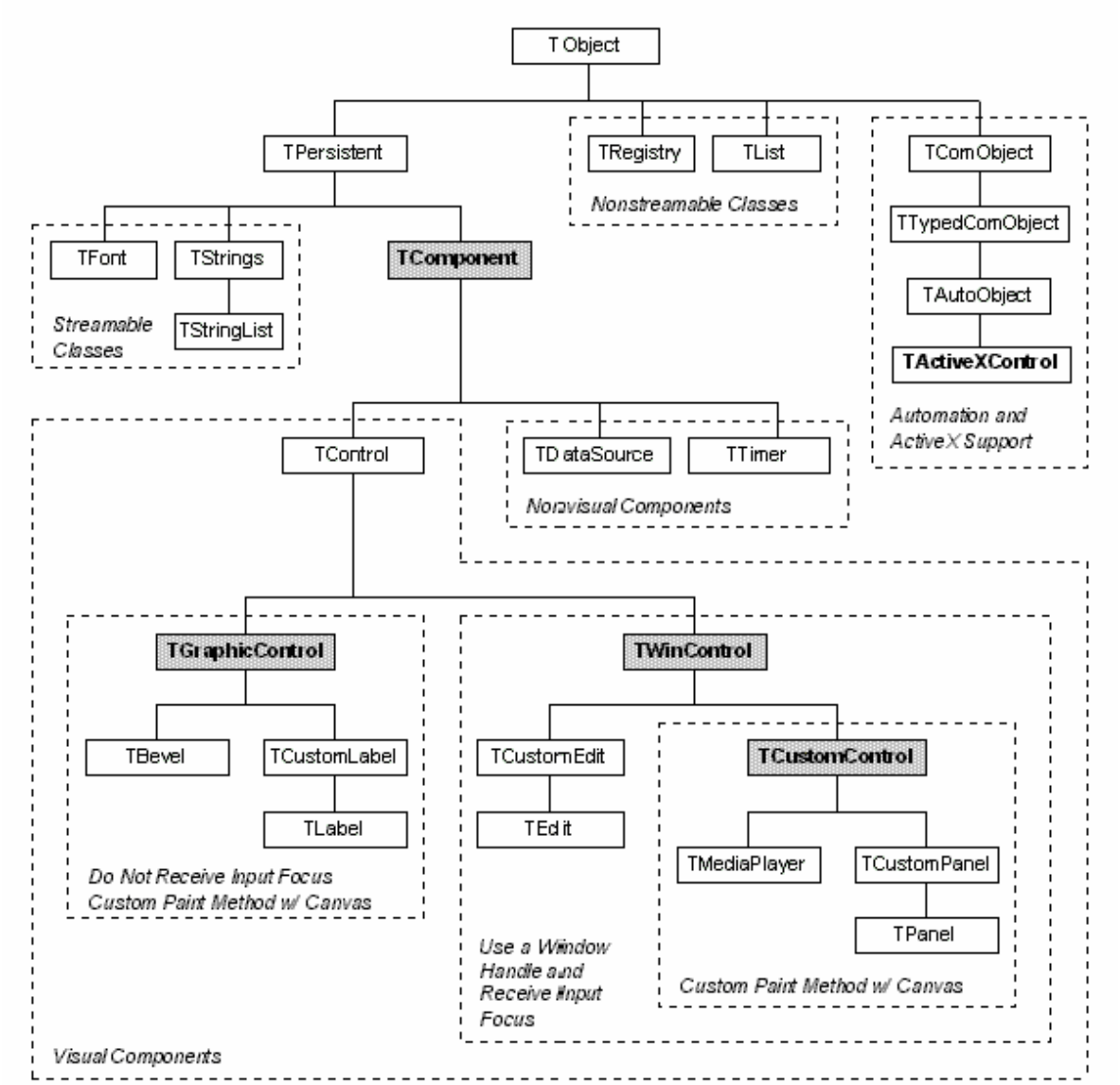


Figura 21. Estrutura base de Classes da VCL.

Como todos componentes são descendentes da classe TComponent, estes possuem uma arquitetura semelhante e são definidos usando a declaração de classe. Um objeto em *Delphi* pode ter métodos, propriedades e eventos. Métodos são ações que um objeto pode realizar. Propriedades representam os valores contidos no objeto. Eventos são condições para as quais um objeto pode executar um método.

A Figura 22 mostra, por exemplo, a arquitetura de um componente em *ObjectPascal,* com seus métodos, propriedades, e eventos. Em (1) tem-se a declaração da classe *TSkeletonComponent* que herda da classe *TComponent* da arquitetura VCL. Em (2) e (3) são declarados os métodos construtor e destrutor, e em (4) e (5) os métodos Method1 e Event1, todos da classe declarada. Em (6) tem-se a declaração de uma propriedade visível em tempo

de projeto, acessada através dos métodos GetProperty1 e SetProperty1, e que tem seu valor armazenado no atributo FProperty1 (sombreado). Em (7) tem-se a declaração de um evento que tem seu valor armazenado no atributo da classe FOnEvent1 (sombreado). O método Event1 (5) é responsável pelo tratamento do evento declarado.

```
Type
  TSkeletonComponent = class(TComponent)    (1)
   Public
     constructor Create( AOwner : TComponent ); override;     (2)
     destructor Destroy; override;    (3)
     procedure Method1;    (4)
     procedure Event1;    (5)
   Private
     FProperty1 : TPropType;
     function GetProperty1 : TPropType;
     procedure SetProperty1 ( Value : TPropType );
     FOnEvent1 : TEventType;
   Published
     property Property1 : TPropType read GetNewProp write SetNewProp;     (6)
     property OnEvent : TEventType read FOnNewEvent write FOnNewEvent;    (7)
end;
```

Figura 22. Arquitetura de um Componente *Delphi*.

Um projeto no ambiente *Delphi* é representado por um conjunto de arquivos visualizados através do *Project Manage*r. De uma forma geral, um Projeto em *Delphi* é composto de formulários, sendo cada formulário definido por dois arquivos, com extensões **".dfm**" e **".pas**". O arquivo **".dfm**" define as propriedades visuais dos componentes, enquanto o arquivo **".pas**", define uma *Unit* na linguagem *ObjectPascal,* contendo a programação para o formulário. A utilização destes dois arquivos tem por finalidade separar as propriedades visuais, da lógica do processo das ações sobre o formulário. Outros arquivos são gerados automaticamente pelo *Delphi* contendo configurações e recursos do projeto, assim como os objetos e o executável, gerados pelo processo de tradução (compilação e link-edição).

Combinando as idéias do método *Catalysis* de DBC, o ST Draco-PUC, a ferramenta MVCASE, e a linguagem *ObjectPascal*, desenvolve-se uma abordagem para construção e reutilização de componentes de software que será apresentada a seguir.

2.8 Considerações Finais

As bibliografias estudadas foram bastante úteis para definir as idéias sobre este projeto. Baseado nos estudos realizados pode-se definir os seguintes pontos importantes para realização deste projeto, destacando-se:

a) Definição de uma primeira idéia sobre a arquitetura dos componentes. Pelas experiências e resultados com a arquitetura VCL do *Delphi*, decidiu-se adotar uma arquitetura semelhante para construção dos componentes e de suas aplicações.

b) Para a abordagem proposta para a construção e reutilização dos componentes será utilizada uma combinação de padrões de projeto suportados pelo ambiente *Delphi*, como solução para os problemas apresentados, de modo a tornar mais compreensível, flexível, e fácil de desenvolver e manter os componentes e suas aplicações. É importante o uso de padrões neste projeto de pesquisa para se obter a documentação dos componentes.

c) A Ferramenta MVCASE e o Sistema de Transformação Draco-PUC são os principais mecanismos utilizados para a implementação dos métodos na Linguagem *ObjectPascal* e para a geração do código *ObjectPascal (Delphi)*, respectivamente, tanto para os componentes construídos quanto para a criação das aplicações que reutilizam os componentes. A Ferramenta MVCASE provê técnicas gráficas e textuais para especificação dos componentes e de suas aplicações, em um alto nível de abstração, e suporta técnicas para persistência. A partir das especificações UML, modeladas na ferramenta MVCASE, o Sistema de Transformação Draco-PUC irá gerar o código na linguagem *ObjectPascal,* por meio de transformações. Para suportar a geração do código são utilizados dois domínios denominados *MDL (Modeling Domain Language)* [Fukuda, 2000] e *ObjectPascal* [Moraes, 2001].

Capítulo 3

Uma Abordagem para Construção e Reutilização de Componentes de Software com Implementação em Delphi

3.1 Visão Geral

Á medida que os sistemas de software se tornam mais complexos, métodos, técnicas e ferramentas que suportem todo o Processo de Desenvolvimento de Software (PDS), desde a identificação dos requisitos até sua implementação, vêm ganhando destaque o Desenvolvimento Baseado em Componentes (DBC). O DBC caracteriza-se por enfocar tanto o desenvolvimento dos componentes de um domínio do problema, como sua reutilização nas aplicações deste domínio.

Embora existam métodos como *UML Components* [Cheesman, 2000], *Catalysis* [Catalysis, 2001], e outros que abordem o DBC, ainda há carência de métodos, técnicas e ferramentas que suportem todo o ciclo de vida do DBC. Esta carência ainda é maior quando se trata de ambientes integrados que tenham ferramentas e técnicas para suportar não apenas a especificação e projeto interno dos componentes, mas também sua implementação e reutilização.

Assim, motivado pelas idéias de construção e reutilização de componentes de software, e com base nos principais conceitos estudados, este projeto apresenta uma Abordagem para Construção e Reutilização de Componentes de Software. A abordagem reúne diferentes conceitos, para suportar todo o processo de Construção e Reutilização de Componentes de Software.

A Figura 23 resume, num Diagrama SADT [1] (*Structured Analysis and Design Technique*) [ROSS, 1977], a idéia proposta. A abordagem parte do estudo do conhecimento de um domínio do problema e produz diferentes artefatos de software que constituem o Projeto dos Componentes e suas implementações. Numa segunda etapa parte-se dos requisitos das aplicações para obter seus projetos e implementações, com reuso dos componentes produzidos na primeira etapa.

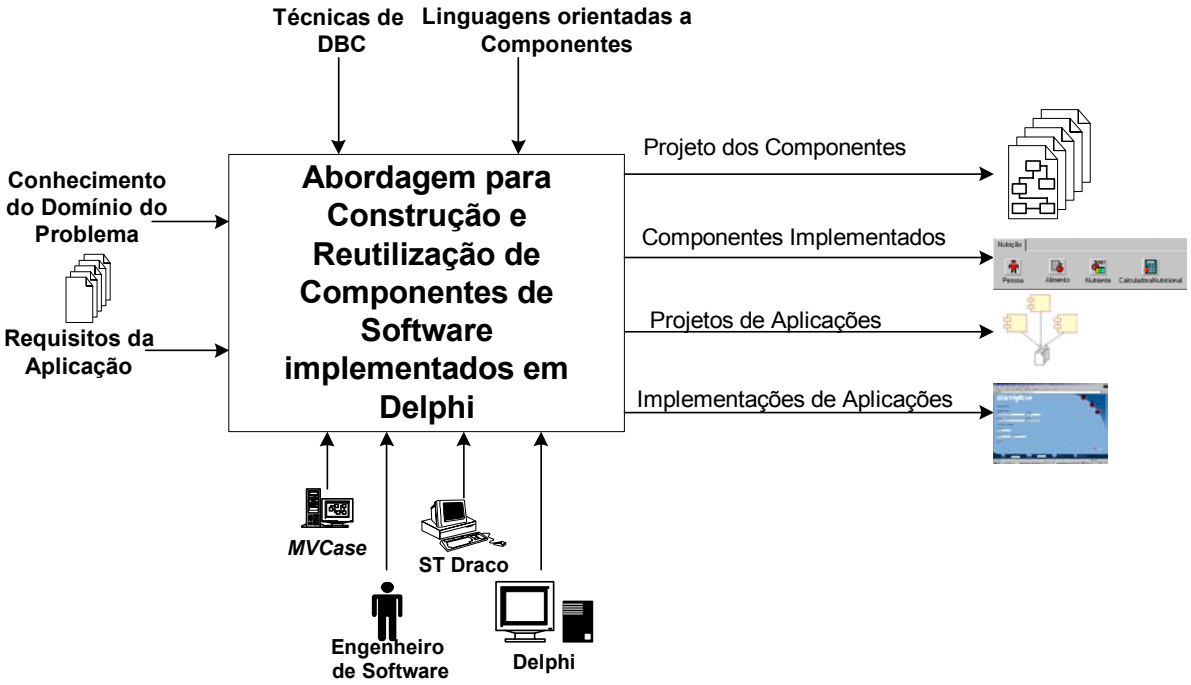
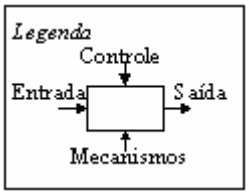


Figura 23 – Abordagem proposta

Os principais mecanismos de execução que auxiliam o engenheiro de software na abordagem proposta são: a ferramenta MVCASE, o Sistema de Transformação Draco-PUC, e o ambiente *Delphi*. A abordagem utiliza técnicas de DBC e Linguagem Orientada a Componentes.

Um conjunto de atividades foi definido para execução da abordagem proposta, que contribui para melhorar o Processo de Desenvolvimento de Software (PDS), principalmente aumentando o grau de reutilização de software e facilitando a sua manutenção. A abordagem,



[1] Legenda *SADT* (*Structured Analysis and Design Technique)*

enfocando o reuso, evita a redundância e melhora a segurança encapsulando nos componentes grande parte de código das aplicações de um domínio. Segue-se uma apresentação detalhada de cada etapa da abordagem.

3.2 Construção e Reutilização de Componentes de Software

A abordagem está dividida em duas grandes etapas: *Construir Componentes* e *Reutilizar Componentes*, conforme mostra a Figura 24. Na primeira etapa, *Construir Componentes*, parte-se do estudo do conhecimento do domínio do problema e obtêm-se Projeto dos Componentes e suas implementações. Na segunda etapa são desenvolvidas as aplicações que reutilizam os componentes construídos na primeira etapa.

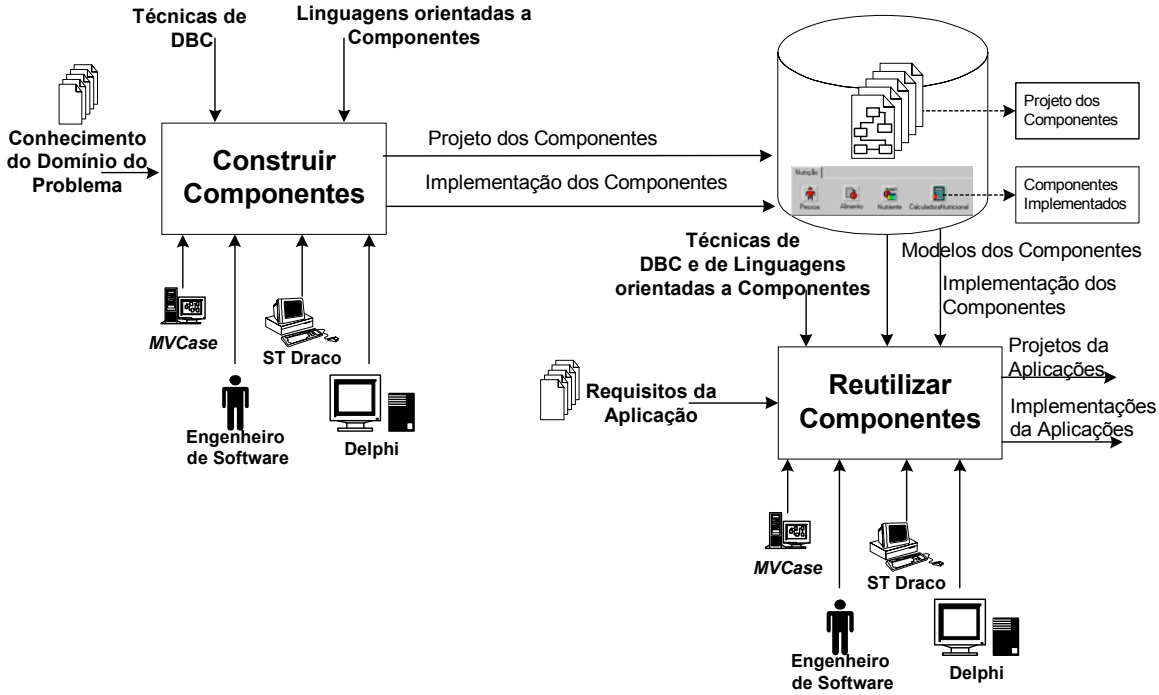


Figura 24 – *Construir e Reutilizar Componentes*

A construção dos componentes de um domínio do problema é realizada em três grandes passos: *Modelar Componentes, Gerar Código dos Componentes e Instalar Componentes*, conforme mostra a Figura 25.

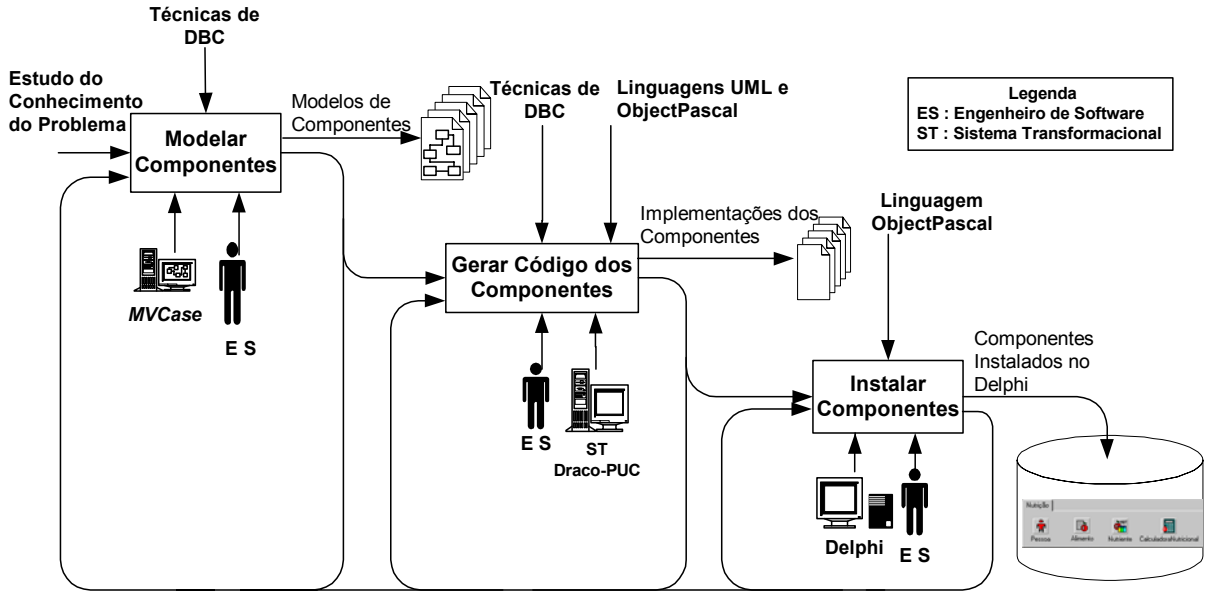


Figura 25 – Passos para *Construir Componentes*

Num primeiro passo faz-se a modelagem dos componentes para obter os projetos dos componentes usando técnicas de DBC. Esta etapa é realizada pelo engenheiro de software com o apoio da ferramenta MVCASE. Estudos sobre o conhecimento do domínio do problema são realizados para identificar os requisitos do domínio do problema. Diferentes fontes, incluindo documentações, entrevistas, reuniões e outras, podem ser utilizadas para identificar os requisitos e obter os conhecimento do domínio do problema.

No segundo passo faz-se a geração de código dos componentes em uma linguagem orientada a componentes. Para este projeto adotou-se a Linguagem *ObjectPascal*, do *Delphi* [Borland, 2001], para implementação dos componentes. O Sistema de Transformação Draco-PUC é utilizado para gerar códigos dos componentes a partir dos seus projetos orientados a componentes.

Num terceiro passo faz-se a instalação dos componentes, no ambiente *Delphi*, disponibilizando-os para reuso das aplicações.

Em qualquer dos passos pode-se retornar aos passos anteriores para corrigir possíveis erros que vão desde uma especificação até uma compilação incorreta dos pacotes de componentes.

Uma vez construído os componentes de um determinado domínio, pode-se reutilizá-los no desenvolvimento de aplicações deste domínio, conforme mostra a Figura 26.

A reutilização dos componentes do domínio de cardiologia é realizada em quatro grandes passos: *Modelar Aplicação, Gerar Código da Aplicação, Executar Aplicação e Refinar Aplicação*, conforme mostra a Figura 26.

O reuso pode ocorrer tanto na modelagem das aplicações como na geração dos seus códigos. Na modelagem, os componentes são importados na ferramenta MVCASE que disponibiliza os Modelos de Componentes para reuso na modelagem das aplicações. Na geração de código da aplicação reutiliza-se o código dos componentes implementados.

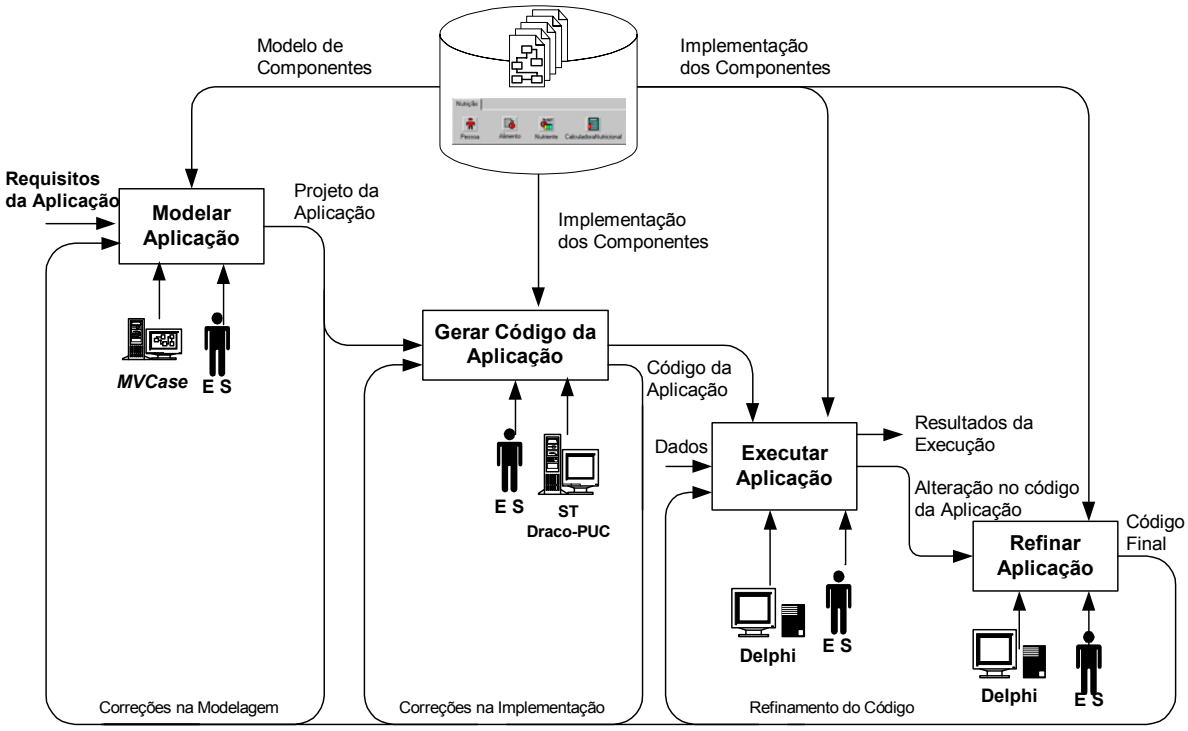


Figura 26 – Passos para *Reutilizar Componentes*

Num primeiro passo faz-se a modelagem da aplicação a partir dos requisitos da aplicação, considerando os modelos dos componentes disponíveis para reuso. Este passo, realizado na MVCASE, resulta no Projeto da Aplicação.

No segundo passo é gerado o código da aplicação, reutilizando o código dos componentes construídos. O Sistema de Transformação Draco-PUC é utilizado para gerar códigos das aplicações a partir dos seus projetos orientados a componentes.

Em seguida o engenheiro de software executa a aplicação a partir do seu código gerado, no ambiente *Delphi*. Neste passo entra-se com dados que permitem testar a aplicação construída. Os resultados da execução servem para verificar se os requisitos da aplicação foram atendidos.

Concluindo a etapa de reutilização dos componentes o engenheiro de software refina a aplicação considerando os requisitos não funcionais, no ambiente *Delphi*. Neste passo é

refinado o código da aplicação para atender requisitos que incluem o tratamento de exceções, interfaces GUI (*Graphic User Interface*) e acesso a banco de dados.

Da mesma forma que na construção dos componentes, pode-se retornar aos passos anteriores para corrigir possíveis erros de modelagem, implementação ou de execução.

Segue-se uma apresentação mais detalhada dos passos de cada etapa da Abordagem, iniciando com a construção de componentes.

Capítulo 4

Construção de Componentes

Para maior clareza e facilitar o entendimento da construção dos componentes de um domínio do problema, segue uma apresentação mais detalhada dos passos: *Modelar Componentes, Gerar Código dos Componentes e Instalar Componentes*

4.1 Modelar Componentes

O passo *Modelar Componentes* é dividido em três passos: *Definir Domínio do Problema, Especificar Componentes e Projetar Componentes*, conforme mostra a Figura 27.

As linhas horizontais tracejadas separam os passos da modelagem dos componentes, segundo os níveis de Abstração do método *Catalysis*: Domínio do Problema, Especificação dos Componentes, Projeto Interno dos Componentes. Embora a abordagem não siga exatamente o método *Catalysis*, grande parte de seus conceitos e técnicas são adotados na abordagem.

No passo *Definir Domínio do Problema*, obtém-se as especificações UML do domínio do problema, a partir estudo do conhecimento do domínio do problema.

No passo *Especificar Componentes* obtém-se os componentes especificados usando técnicas de DBC.

No passo *Projetar Componentes* faz-se o projeto interno dos componentes, usando técnicas de DBC do método *Catalysis* [D´Souza, 1999].

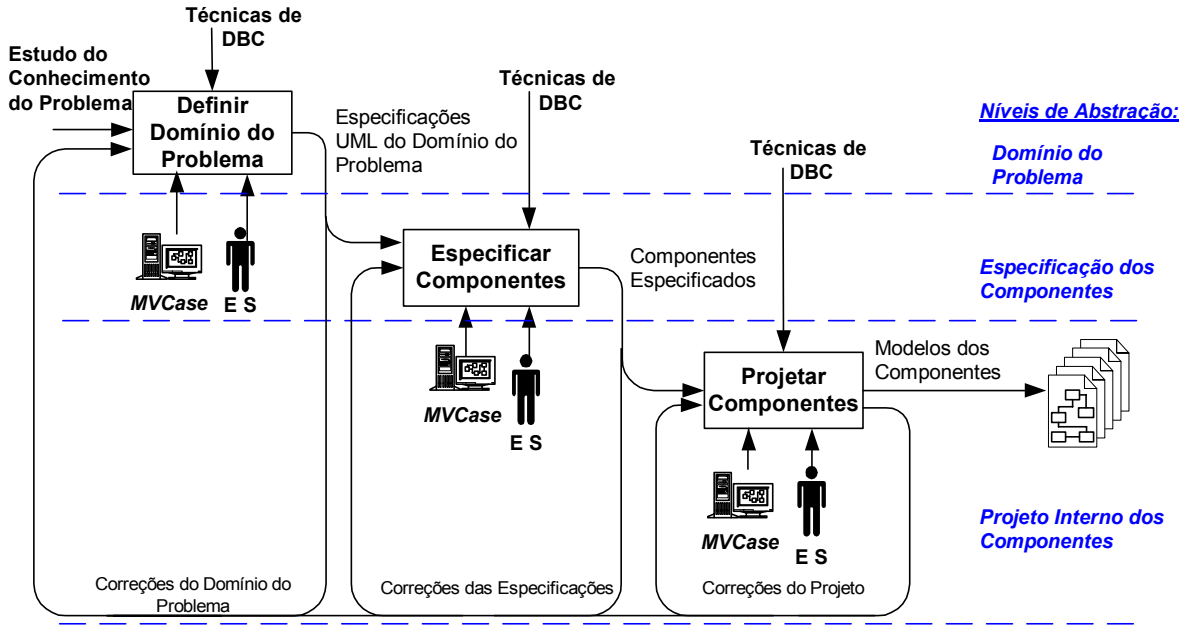


Figura 27 – Passos para *Modelar Componentes*

Segue-se a apresentação detalhada dos passos para modelagem dos componentes.

4.1.1 Definir Domínio do Problema

Para melhor entendimento da abordagem, será utilizado o "Domínio de Cardiologia" para exemplificar os passos da construção dos componentes. A utilização do domínio de cardiologia tem como base as experiências deste pesquisador no desenvolvimento de um sistema de cardiologia com 320 classes, e o desenvolvimento de outros softwares para a área de saúde [Moraes, 2001], [Moraes, 2001a], [Moraes, 2002c], [Moraes, 2002d], [Moraes, 2004].

No primeiro passo, *Definir Domínio do Problema***,** é dado ênfase no entendimento e no conhecimento do problema, especificando-se "*o quê*" os componentes devem atender para solucionar o problema.

Inicialmente, são elicitados os conhecimentos de um domínio do problema, usando técnicas como *storyboards* ou *mind-maps* [D´Souza, 1999], visando representar as diferentes situações e cenários do domínio do problema, e capturar o contexto e a abrangência do domínio. A Figura 28 mostra em um *mind-map* o contexto do domínio do problema, dividido em quatro áreas de assunto (*Test, Surgery, Appointment e Hemodynamic*), facilitando o entendimento relacionado ao domínio de cardiologia.

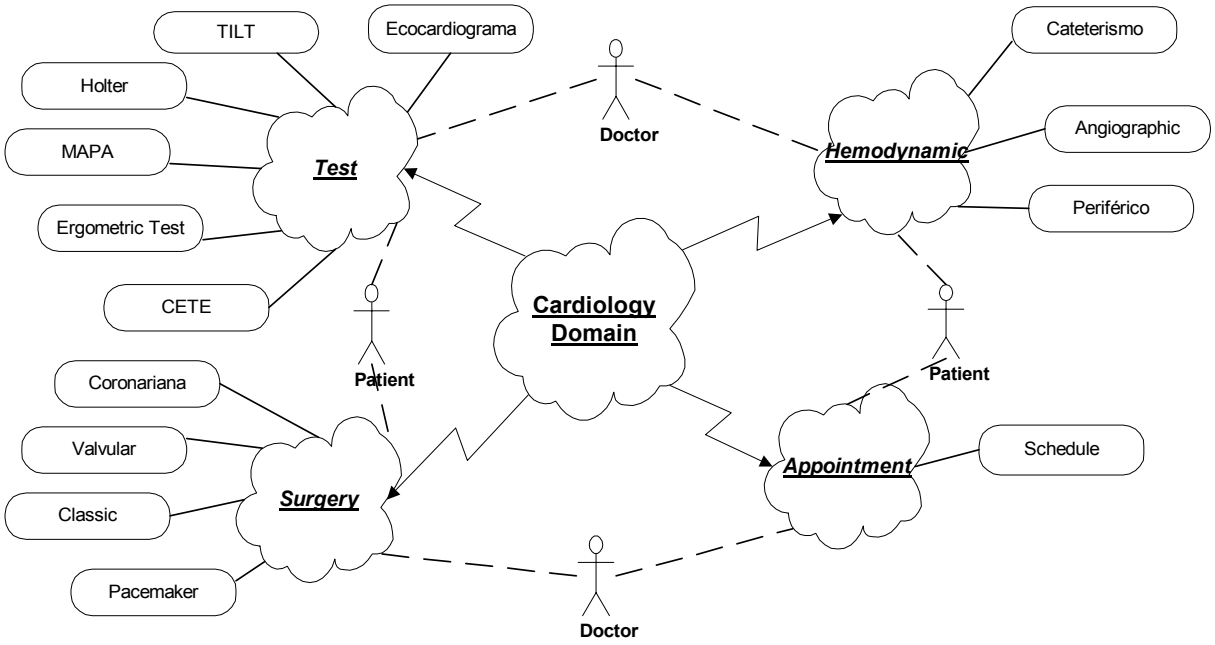


Figura 28– ***Mind-map - Domínio do Problema***

Os requisitos identificados são especificados pelas regras de negócios que expressam o entendimento do domínio do problema, representado em um Modelo de Tipos de Ações, refletindo os processos existentes.

Por exemplo, no caso do domínio de Cardiologia, partiu-se de um levantamento de requisitos realizado para desenvolver o Sistema de Cardiologia (SisCardio) do Instituto do Coração de Marília, implantado em 1997. Na ocasião, profissionais e médicos da área de cardiologia e um sistema de software do Instituto do Coração de Marília (ICM), o SisCardio, foram as principais fontes para identificação dos requisitos. A Figura 29 mostra parte do Modelo de Entidade-Relacionamento (MER) do SisCardio, obtida através da ferramenta de modelagem de dados do SGBD relacional SQL Server 2000, que serviu de base para a definição dos componentes persistentes da presente abordagem. Reuniões, entrevistas, estudo de sistemas legados e observações de cenários de uso do sistema foram empregados no levantamento e identificação dos requisitos. Com a utilização do sistema surgiram novos requisitos e, ao longo do tempo, foram necessárias atualizações para acompanhar mudanças de tecnologias. Todas estas experiências foram importantes para conhecer o domínio de Cardiologia, facilitando a construção dos componentes [Moraes, 2002], [Moraes, 2002a], [Moraes, 2002b], e [Moraes, 2003].

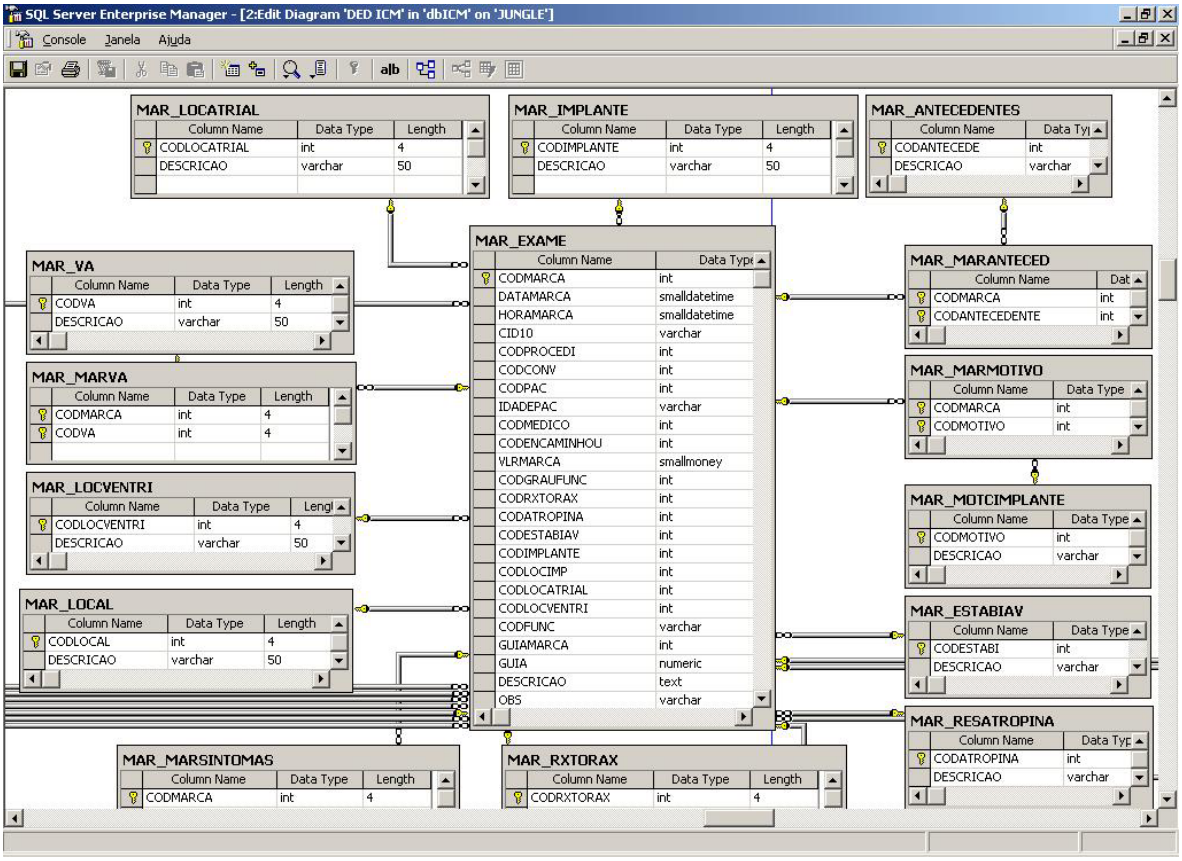


Figura 29– *Modelo de Entidade Relacionamento do SisCardio*

Pela Abordagem proposta, sugere-se que o desenvolvedor proceda da mesma forma com outros domínios, utilizando técnicas já conhecidas para identificação de requisitos de software, complementadas com outras mais recentes apresentadas nos métodos *UML Components*, *Catalysis* e outros. Dentre as técnicas de Catalysis usadas nesta fase destacam-se os modelos de *Mind-map e Storyboard*.

Em seguida, os requisitos identificados são especificados em Modelos de Tipos de Ações, representando os processos de negócio, os papéis dos atores e as ações entre os objetos participantes. Os objetos são usados para modelagem e auxiliam no entendimento do domínio do problema, tornando-o mais claro. Um tipo identificado é uma visão abstrata de um objeto e descreve o comportamento do objeto através das ações. Finalmente, os Modelos de Tipos de Ações são refinados em Modelos de Casos de Uso, visando representar as interações entre os objetos.

A Figura 30 resume o primeiro passo para a modelagem dos componentes, onde um *mind-map*, definido na identificação do problema, é especificado num Modelo de Tipos de Ações, e, posteriormente, particionado e refinado em Modelos de Casos de Uso, visando diminuir a complexidade e melhorar o entendimento do domínio do problema.

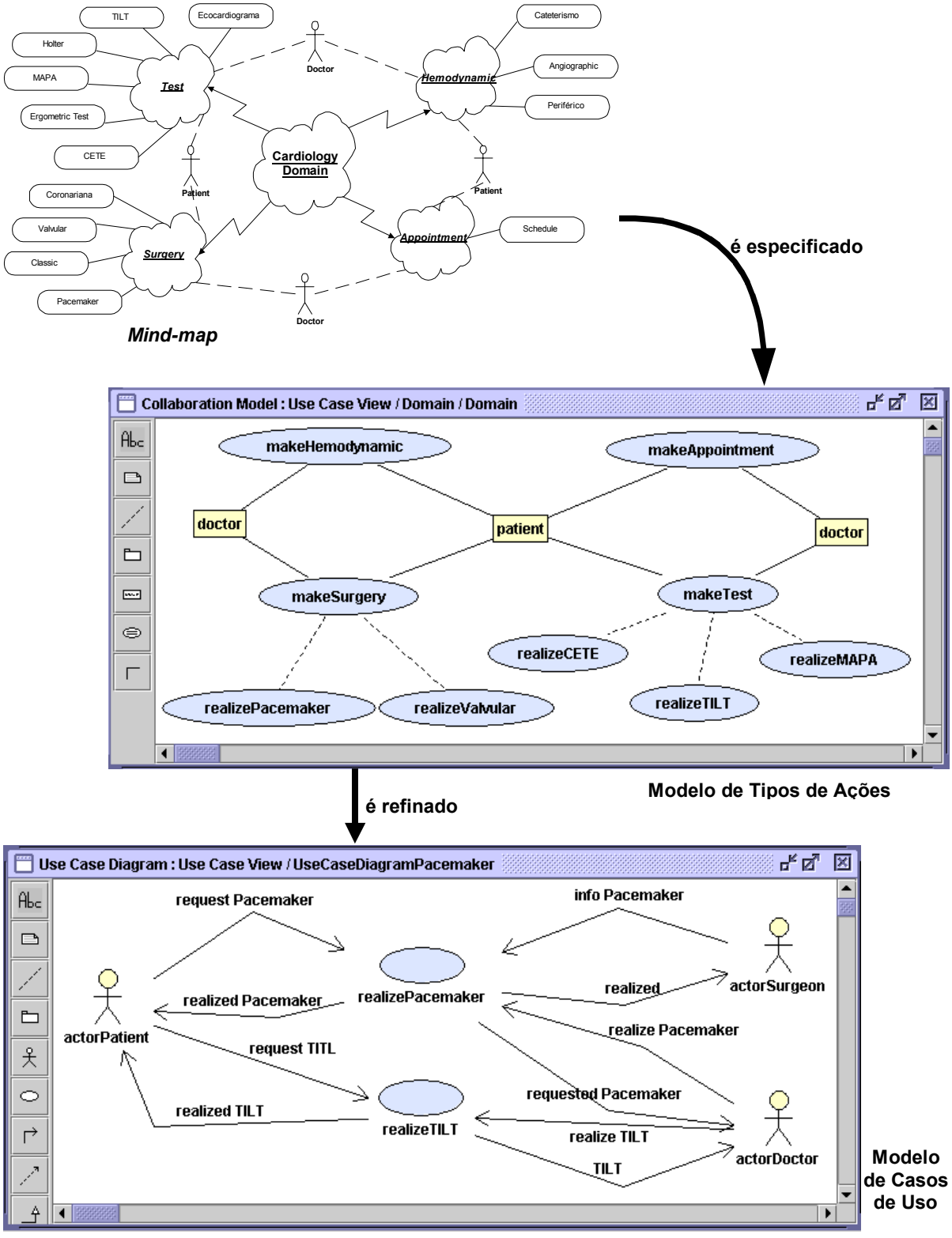


Figura 30- ***Definir Domínio do Problema***

Uma vez definido o domínio do problema o engenheiro de software inicia outro passo da Abordagem que consiste em Especificar os Componentes.

4.1.2 Especificar Componentes

No passo *Especificar Componentes***,** semelhante ao segundo nível do método *Catalysis*, são descritos os comportamentos externos dos componentes de uma forma não ambígua. Enquanto no passo anterior a preocupação do engenheiro de software é com a identificação dos requisitos e atores, agora o enfoque é para a sua análise e modelagem.

Esse passo tem início com o refinamento mais detalhado dos modelos do domínio do problema. Outros modelos podem ser usados como o de *Snapshot* que auxilia no refinamento dos Tipos e seus Atributos. Assim, o engenheiro de software refina os modelos do passo anterior, visando obter as especificações dos componentes. Por exemplo, a Figura 31 mostra um *Snapshot*, que retrata um conjunto de objetos e os valores de alguns de seus atributos em um determinado ponto do tempo. O Snapshot é um diagrama de instâncias que descreve o comportamento dos objetos, no caso os envolvidos na ação *realizePacemaker*.

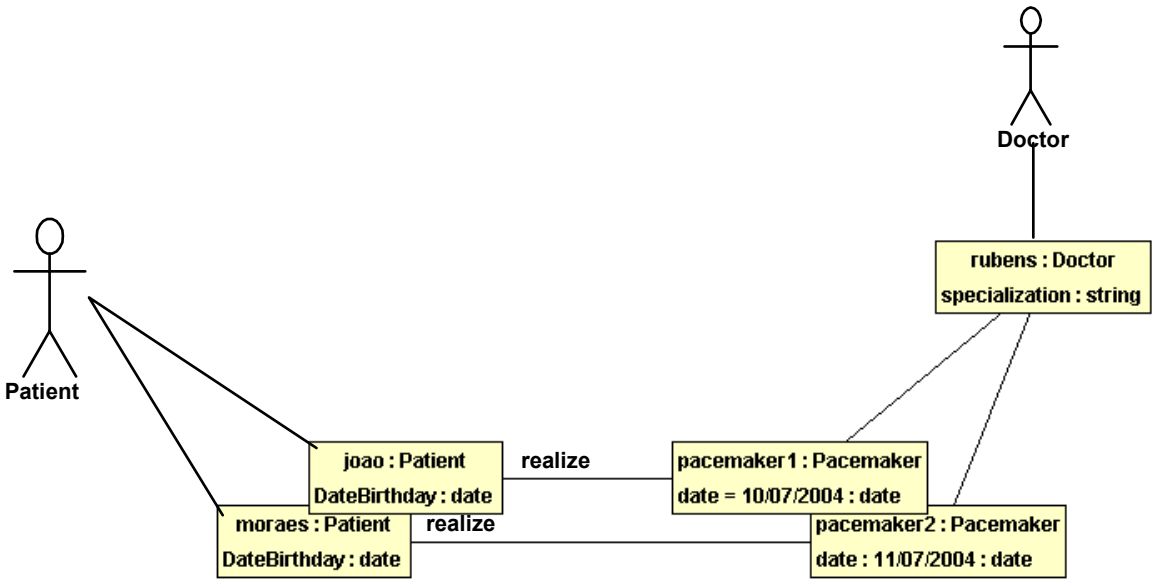


Figura 31– *Snapshot* usado para determinar os atributos dos objetos

Assim, refinando o Modelo de Tipos de Ações e generalizando os *Snapshots*, tem-se os Modelos de Tipos para os diferentes casos de uso identificados no domínio do problema. Modelo de Tipos vem do refinamento dos Modelos de Casos de Uso do passo anterior e dos Diagramas de Instâncias (*Snapshot*), descrevendo o comportamento dos objetos.

Os tipos são relacionados através de associações, agregações ou herança, com as respectivas cardinalidades, que expressam as ocorrências mínimas e máximas de objetos participantes de um relacionamento. Conforme mostra a Figura 32, os Modelos de Tipos contêm um conjunto de instâncias dos *Snapshots*. Neste modelo além dos atributos, têm-se representadas as ações identificadas no modelo de tipos de ações e refinadas em modelos de casos de uso. Embora se possam especificar detalhes, como por exemplo se um atributo é *String* ou *Integer*, e as cardinalidades dos relacionamentos entre os Tipos, a preocupação ainda não deve ser com detalhes físicos da implementação.

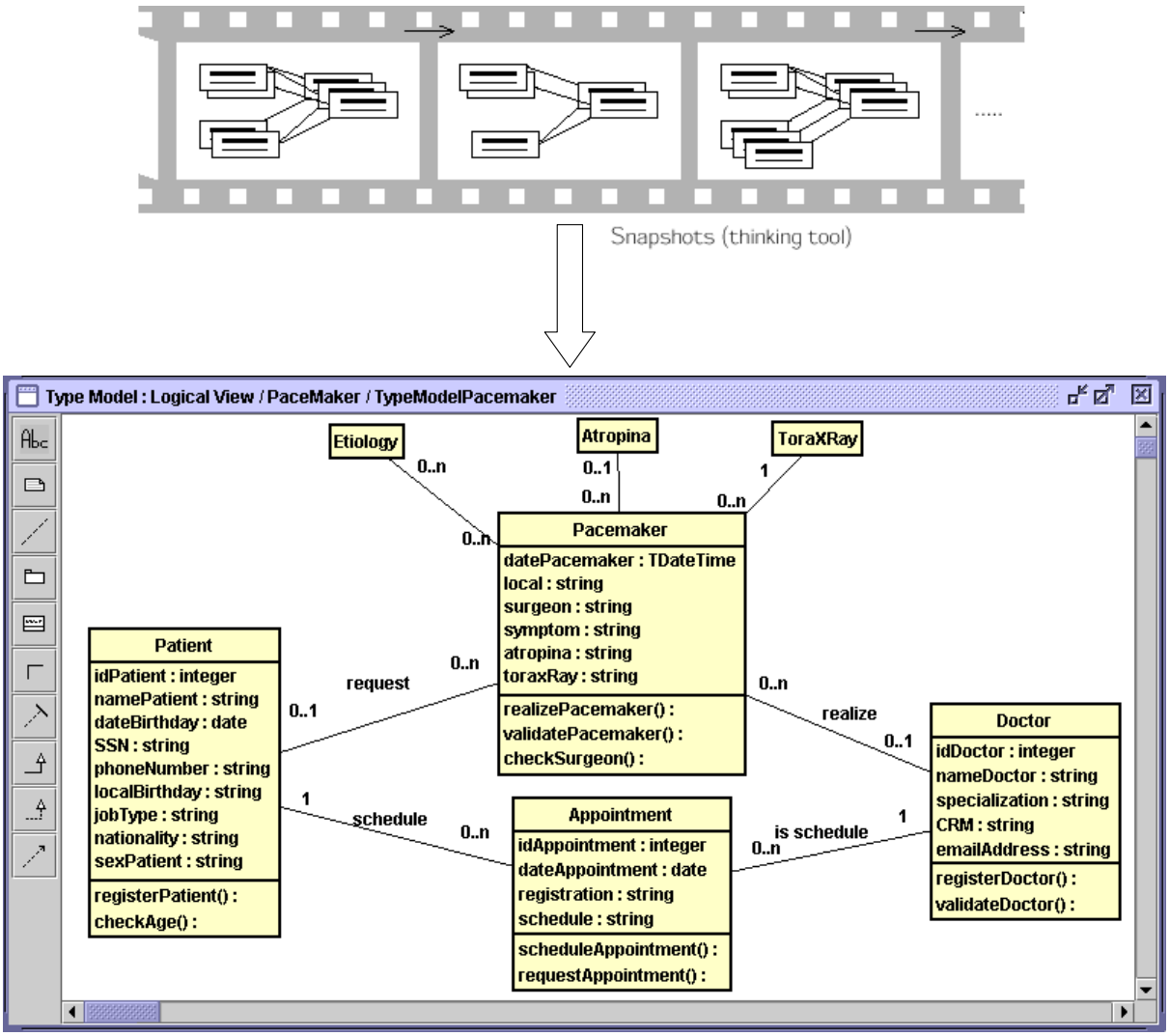


Figura 32– *Snapshot* e Modelo de Tipos do passo *Especificar Componentes*

Outra técnica usada neste passo é a do Modelo de Colaboração [D´Souza, 1999]. Uma Colaboração é representada por um conjunto de ações entre objetos. Ações estas definidas por meio de um modelo de tipos comuns entre os objetos envolvidos. A Figura 33 mostra uma

colaboração, denominada **Make Cardiology Surgery**, que realiza uma cirurgia . O modelo de colaboração descreve um conjunto de duas ações, *realizePacemaker* e *makeAppointment*, entre *Patient* e os tipos *Pacemaker* e *Appointment*. No caso a ação *realizePacemaker* é uma especialização da ação *makeSurgery*.

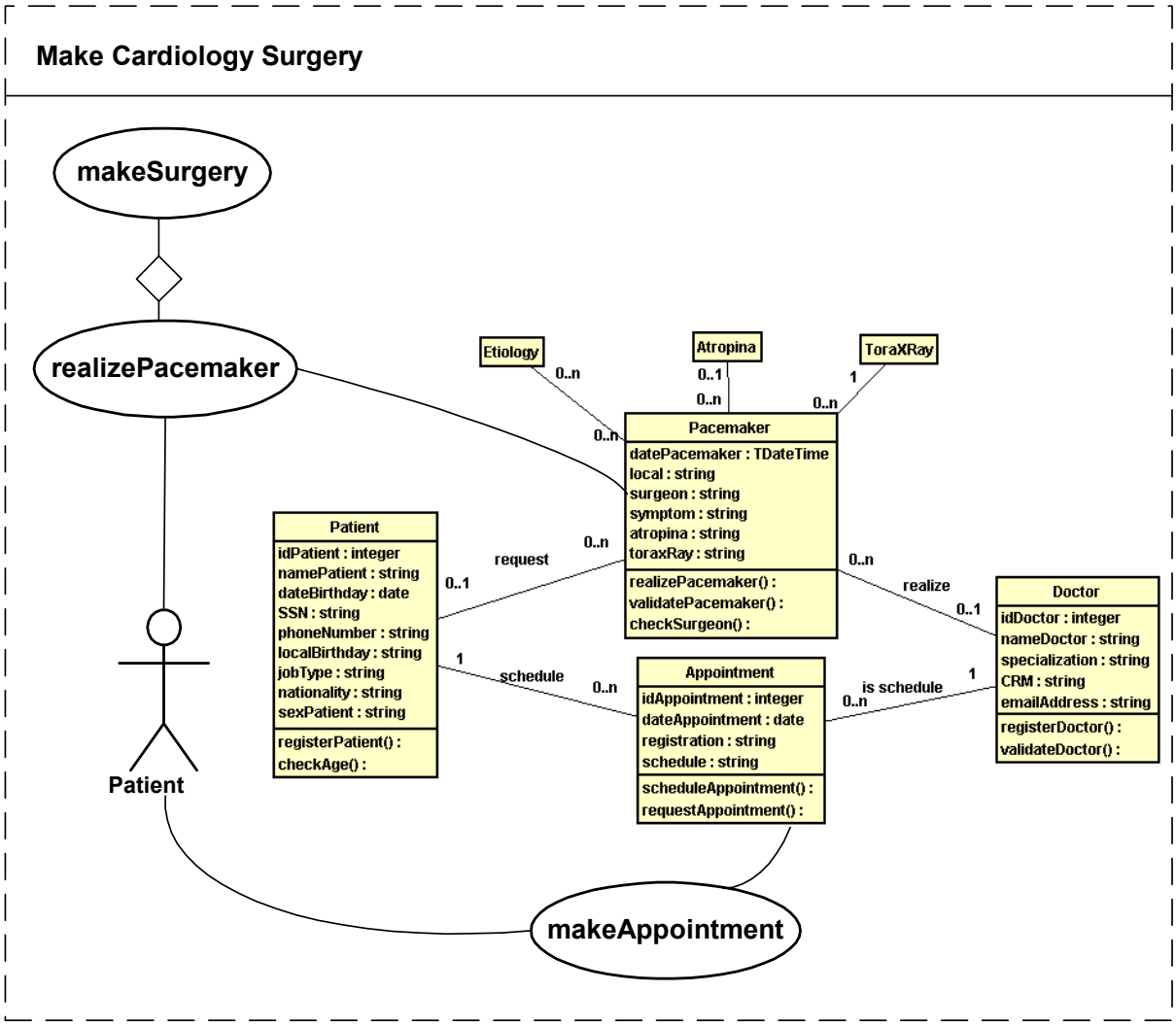


Figura 33– Colaboração que realiza uma cirurgia abstrata

Outro modelo especificado nesta etapa é o Modelo de Interações, que refina o comportamento dos casos de uso. O modelo de interações pode ser representado em diagrama de seqüência ou num grafo que relaciona os objetos. A Figura 34 mostra um modelo de interação usando o diagrama de seqüência que detalha o comportamento do caso de uso *realizePacemaker*.

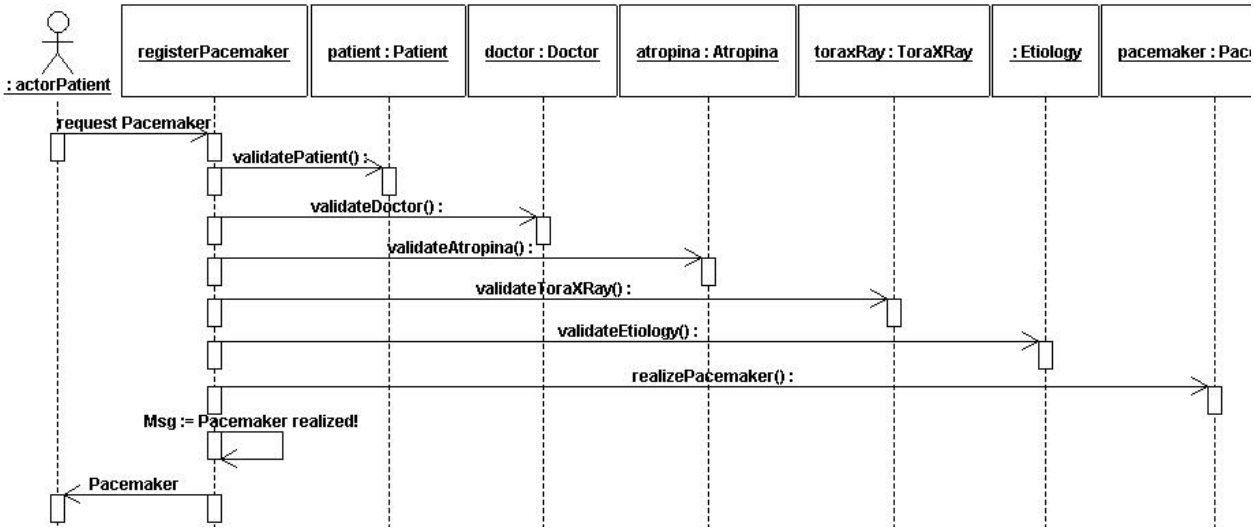


Figura 34– Modelo de Interações usando Diagrama de Seqüência

Baseado no modelo de interação pode-se documentar textualmente os casos de uso. Por exemplo, a Figura 35 mostra a documentação dos cursos normal e alternativos do caso de uso *realizePacemaker*.

Caso de Uso: *realizePacemaker*
Descrição: Este caso de uso trata da realização de um implante de Marcapasso em um Paciente.
Ator: *Patient*

Curso Normal:

1. *Patient* solicita realização de um implante de Marcapasso.
2. Sistema verifica se existe *Patient* informado.
3. Sistema verifica se existe *Doctor* informado.
4. Sistema verifica se existe *Atropina* informada.
5. Sistema verifica se existe *ToraxRay* informado.
6. Para cada *Etiology* informada, sistema verifica se existe *Etiology* informada.
7. Sistema registra uma instância para o Tipo *Pacemaker*.
8. Sistema emite mensagem que o Marcapasso foi registrado. Msg = *Pacemaker realized!*.

Curso Alternativo:

2. Caso *Patient* não exista.
2.1 Sistema emite mensagem informando que *Patient* não existe, e encerra o caso de uso.

3. Caso *Doctor* não existe.
3.1 Sistema emite mensagem informando que *Doctor* não existe, e encerra o caso de uso.

4. Caso *Atropina* não exista.
4.1 Sistema emite mensagem informando que *Atropina* não existe, e encerra o caso de uso.

5. Caso *ToraxRay* não exista.
5.1 Sistema emite mensagem informando que *ToraxRay* não existe, e encerra o caso de uso.

6. Caso *Etiology* não exista.
6.1 Sistema emite mensagem informando que *Etiology* não existe, e encerra o caso de uso.

Figura 35– Curso Normal e Alternativo do caso de uso *realizePacemaker*

No refinamento dos casos de uso, obtendo-se os modelos de interações, pode-se observar a adição de novas ações no Modelo de Tipos. Pro exemplo, a Figura 36 mostra uma nova ação, *validatePatient*( ), adicionada ao tipo *Patient*.

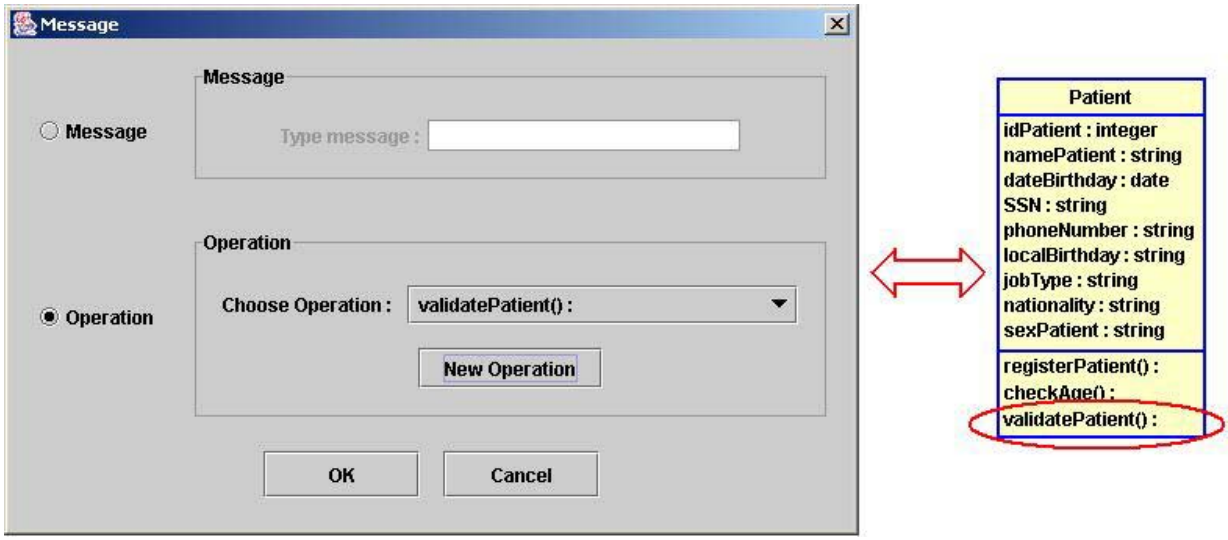


Figura 36– Nova Ação adicionada ao Tipo *Patient*

Em resumo, as atividades deste passo, realizadas pelo engenheiro de software, na ferramenta MVCASE, compreendem a especificação do:

- Modelo de Tipos, obtido da generalização dos *Snapshots*;
- Modelo de Colaborações entre os objetos envolvidos;
- Modelos de Interações, representados pelos diagramas de seqüência, baseados nos modelos de casos de uso do passo anterior; e
- Documentação dos casos de uso.

Estes Modelos são refinados no próximo passo *Projetar Componentes*, para obter o projeto interno dos componentes.

4.1.3 Projetar Componentes

No passo *Projetar Componentes,* o engenheiro de software realiza o projeto interno dos componentes, conforme o terceiro nível de *Catalysis*. Requisitos não funcionais, como por exemplo os relacionados com a arquitetura dos componentes e a persistência dos objetos, são tratados neste passo.

Inicialmente, os Modelos de Tipos, obtidos no passo anterior, são refinados em Modelos de Classes, levando em consideração a definição dos componentes com suas interfaces. Conforme mostra a Figura 37 as classes denotam a implementação dos tipos modelados no passo anterior. Os comportamentos abstratos das ações são detalhados em mini-especificações, gerando os métodos das classes. Atributos são refinados com seus tipos, tempo de vida e visibilidade, e os protótipos dos métodos são especificados considerando seus parâmetros, tipos de retorno e outros modificadores como, por exemplo, o que define a sua visibilidade.

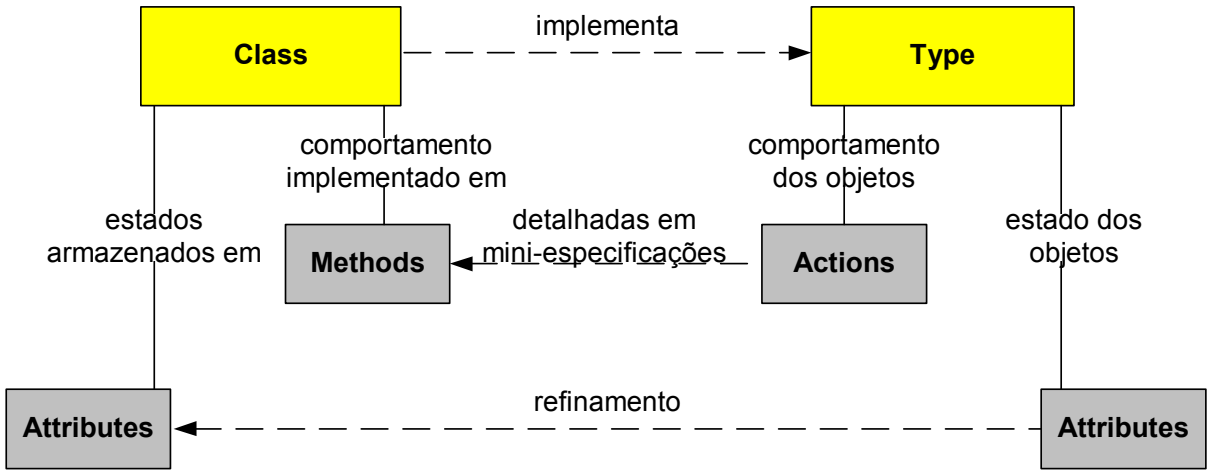


Figura 37– Mapeamento conceitual de Tipos e Classes

Este passo compreende: o Projeto do Modelo de Classes, o Projeto do Modelo de Componentes, e o Projeto do Modelo de Dados.

4.1.3.1 Projeto do Modelo de Classes

Refinamentos nas assinaturas dos métodos e nos atributos das classes podem ser necessários nesta fase.

A Figura 38 mostra parte do Modelo de Classes dos Componentes do domínio de cardiologia, obtido do modelo de tipos do passo anterior. O engenheiro de software utiliza, na ferramenta MVCASE, os *stereotypes* <<*Session*>> e <<*Entity*>> para definir as classes dos componentes que serão transientes e persistentes, respectivamente.

Session define uma classe cujos objetos não são armazenados em banco de dados, existindo apenas durante suas execuções. *Entity* define uma classe cujos objetos são armazenados em banco de dados.

Os Modelos de Interações, representados através dos diagramas de seqüência, são refinados para mostrar detalhes de projeto dos comportamentos dos métodos em cada classe.

Conforme mostra a Figura 38, devido ao grande número de tipos e para uma melhor visualização, no *browser* da ferramenta MVCASE, as classes foram organizadas em pacotes, especificando-se os pacotes do domínio de cardiologia. O engenheiro de software, refinando este modelo, deve definir o *stereotype* <<*Cardiology*>> para os pacotes do domínio de cardiologia.

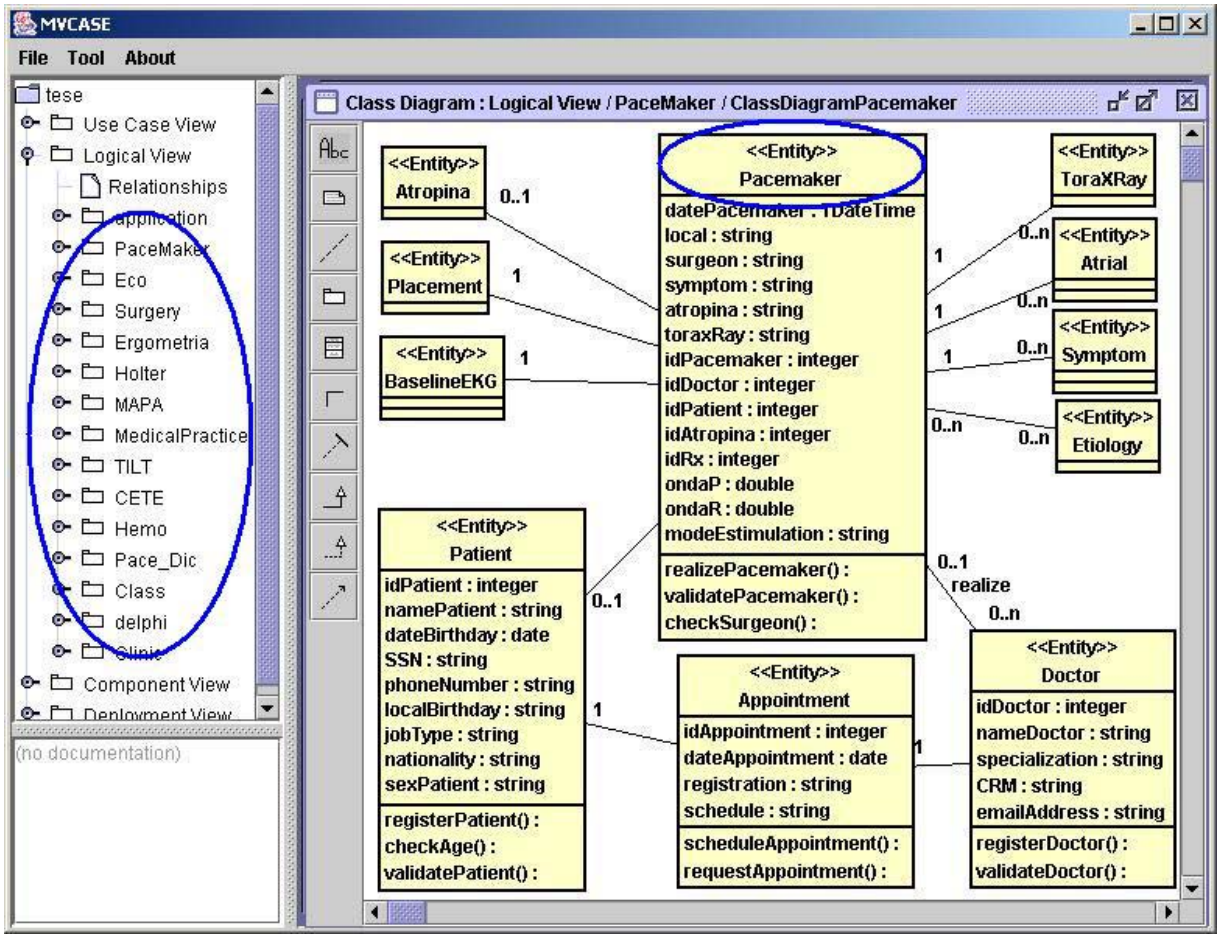


Figura 38- Modelo de Classes obtido do Modelo de Tipos

Nesta fase ainda o engenheiro de software precisa definir os *stereotypes* dos métodos oriundos dos tipos ou para os novos métodos. São utilizados os seguintes *stereotypes*:

- *<<BusinessRule>>* referente às regras de negócio do domínio do problema;
- *<<DelphiConstructor>>* e *<<DelphiDestructor>>* referente ao ciclo de vida dos objetos;
- *<<DelphiPersistence>>* referente aos métodos de persistência dos objetos;
- *<<DelphiFinder>>* que representa os métodos de localização de instâncias; e
- *<<DelphiVirtualFieldGet>>* e *<<DelphiVirtualFieldSet >>* relacionados aos métodos de acesso aos atributos.

Para os atributos das classes deve-se definir a visibilidade dos atributos e os modificadores, utilizando a opção *key* para os atributos que serão as chaves-primárias para as classes do tipo *Entity*.

A Figura 39 mostra a visibilidade do atributo *datePacemaker* definida como *Published.* De uma forma geral, baseado na arquitetura VCL do *ObjectPascal* do *Delphi*, adotada para implementação dos componentes, os atributos e métodos podem ser:

- *PRIVATE*: Acesso privado, isto é, as variáveis e métodos não são acessíveis fora da *Unit* onde a classe está declarada;
- *PUBLIC*: Acesso público, isto é, as variáveis e métodos são livremente acessados por qualquer *Unit*, mesmo aquelas que não contenham a declaração da classe, desde que na cláusula *Uses* da outra *Unit* exista referência à mesma;
- *PROTECTED*: Acesso protegido, isto é, as variáveis e métodos têm visibilidade limitada, ou seja, apenas a classe atual e suas sub-classes têm acesso a estes dados; e
- *PUBLISHED*: Está disponível também em tempo de projeto, pois, representa as características do objeto apresentadas no *Object Inspector*. Normalmente escreve-se nesta parte a propriedade do objeto.

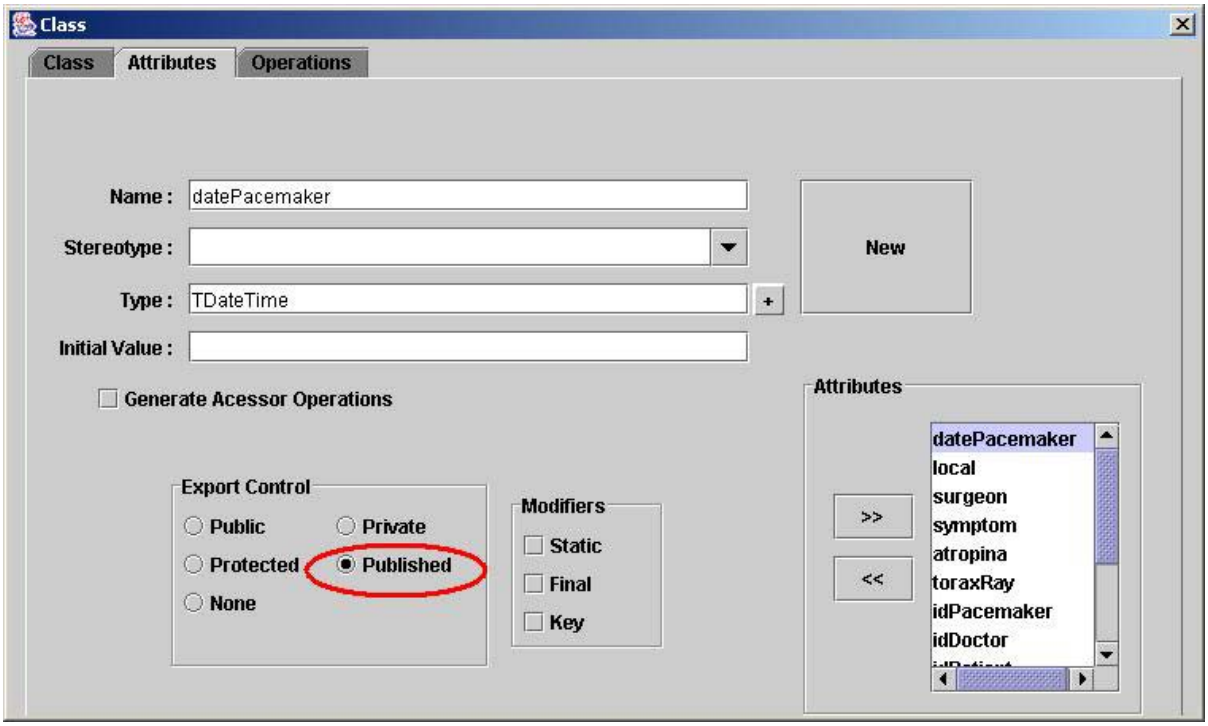


Figura 39– Visibilidade dos atributos

A partir do Modelo de Classes obtém-se automaticamente, na ferramenta MVCASE, uma primeira versão do Modelo de Componentes.

4.1.3.2 Projeto do Modelo de Componentes

Conforme mostra a Figura 40, através da opção *"Transform Classes to Delphi Components (By Stereotype)"* são gerados os componentes do domínio do problema.

Componentes do *ObjectPascal* do *Delphi* são importados, ficando disponíveis para reuso no *browser* da ferramenta MVCASE.

O pacote de componentes da arquitetura *VCL* do *delphi* tem seu *stereotype* definido como <<*Delphi Framework*>>.

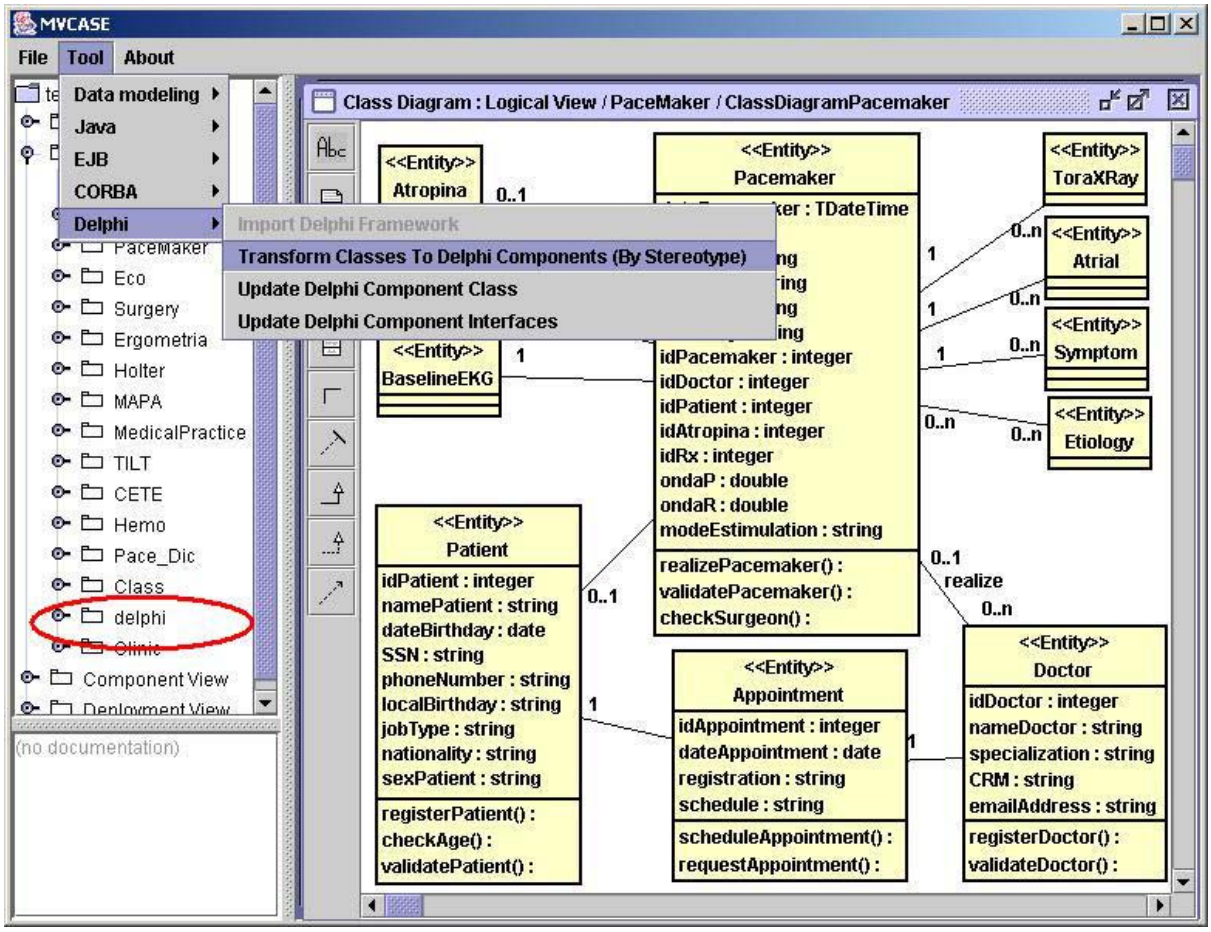


Figura 40– Geração do Modelo de Componentes

A Figura 41 mostra o Modelo de Classes dos Componentes obtido do Modelo de Classes da Figura 40, onde são utilizados os *stereotypes* *<<DelphiTransient>>* e *<<DelphiPersistent>>* para distinguir componentes transientes e persistentes.

Em (1) pode-se observar também que foram gerados automaticamente pela ferramenta MVCASE novos atributos, e em (2) os novos métodos conforme as especificações do modelo de classes. O prefixo "*F*" foi adicionado aos novos atributos gerados para denotar os *Fields* do *ObjectPascal*. Os novos métodos gerados, com prefixo *Get* e *Set*, foram adicionados para acessar os atributos com visibilidade *Private*.

O prefixo "*T*" foi adicionado aos tipos para denotar o *Type* do *ObjectPascal*, que representa uma classe, como no caso da classe *TPacemaker*.

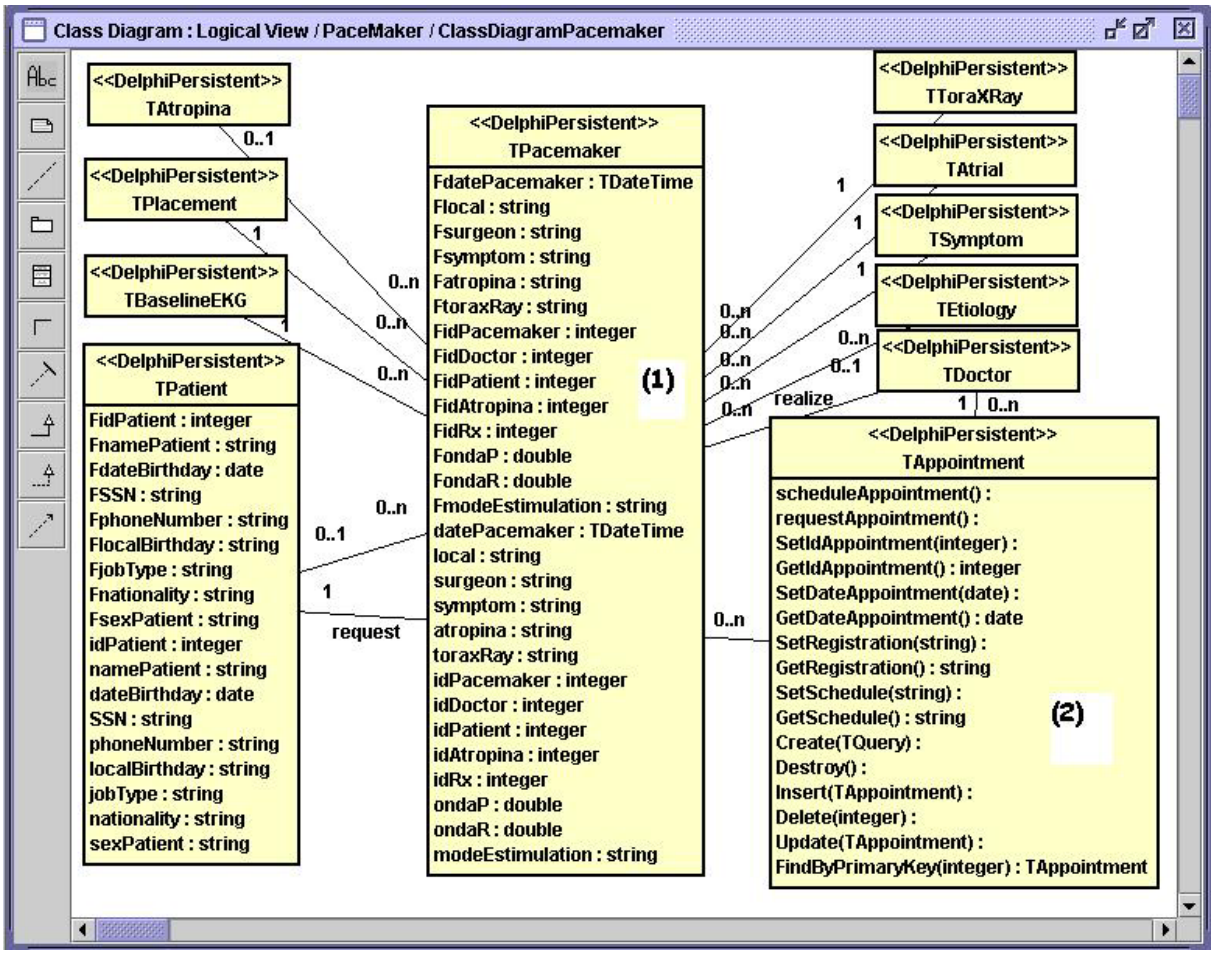


Figura 41- Modelo de Classes dos Componentes do pacote *Pacemaker*

Outro modelo deste passo é o de Componentes, que representa os componentes, com suas interfaces para conexão e suas dependências, semelhante à arquitetura VCL, onde cada componente disponibiliza seus serviços através de duas interfaces, *Designtime* e *Runtime*. Neste modelo as interfaces dos componentes podem ser representadas pelos ícones em forma de círculos. Seguindo a idéia de componentes "*Plug-In*" de *Catalysis*, a conexão entre componentes se dá por meio da interface, e é representada por um relacionamento de dependência.

A Figura 42 mostra o Modelo de Componentes obtido da transformação do Modelo de Classes na ferramenta MVCASE. O componente *TPacemaker* é do tipo *DelphiPersistent* e possui duas interfaces *IDesigntimePacemaker e IRuntimePacemaker,* disponibilizando seus serviços. A conexão entre os componentes *TPacemaker* e *TDoctor*, por exemplo, se dá por meio das interfaces *IDesigntimeDoctor e IRuntimeDoctor,* e é representada por um relacionamento de dependência.

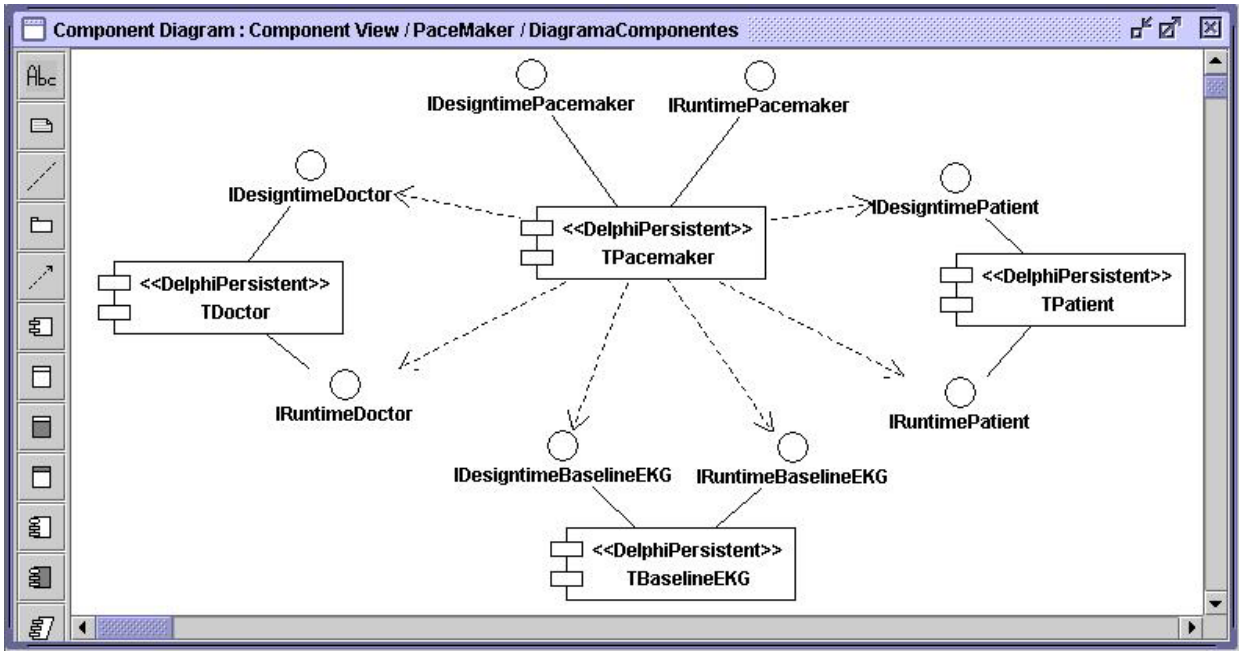


Figura 42- Modelo de Componentes

Conforme mostra a Figura 43, o componente *TPacemaker* tem sua implementação especificada na classe *TPacemaker*, e disponibiliza seus serviços através de duas interfaces *IDesigntimePacemaker e IRuntimePacemaker.*

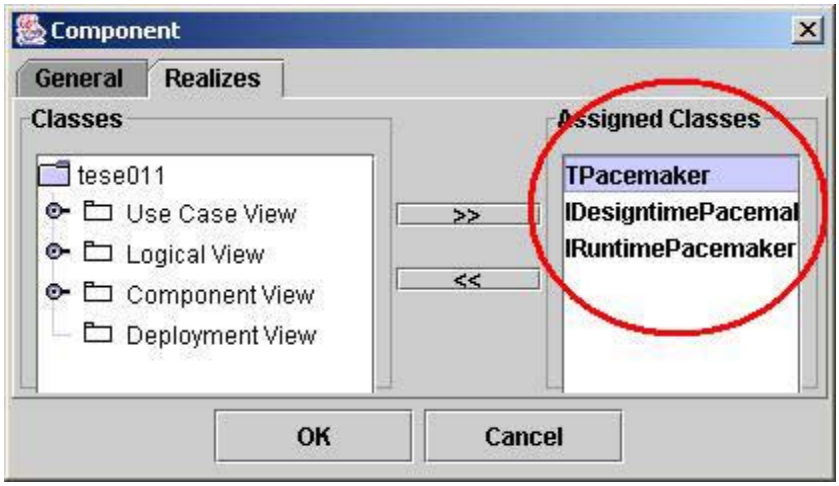


Figura 43– Classe que realiza o componente *TPacemaker*

Propriedades são atributos que determinam o status e o comportamento de um objeto. Sempre que um valor é lido ou alterado em uma propriedade, o valor do objeto é acessado diretamente ou através de um método.

Na Figura 44, por exemplo, mostra com mais detalhe a classe *TPacemaker* que implementa o componente *TPacemaker* com suas interfaces, *IDesigntimePacemaker e IRuntimePacemaker*.

A interface que tem o prefixo "*IDesigntime*" disponibiliza as propriedades e os métodos do componente, no *Object Inspector* do ambiente *Delphi,* em tempo de projeto, como a propriedade *datePacemaker*. E a interface que tem o prefixo "*IRuntime*", contem os métodos de Persistência, de Ciclo de Vida, de localização de instâncias, e Regras de Negócio, disponíveis em tempo de execução, como por exemplo os métodos de persistência: *Insert*( ), *Delete*( ) e *Update*( ), os métodos de ciclo de vida: *Create*( ) e *Destroy*( ), e o método de localização de instância: *FindByPrimaryKey* ( ), e a regra de negócio: *checkSurgeon*( ).

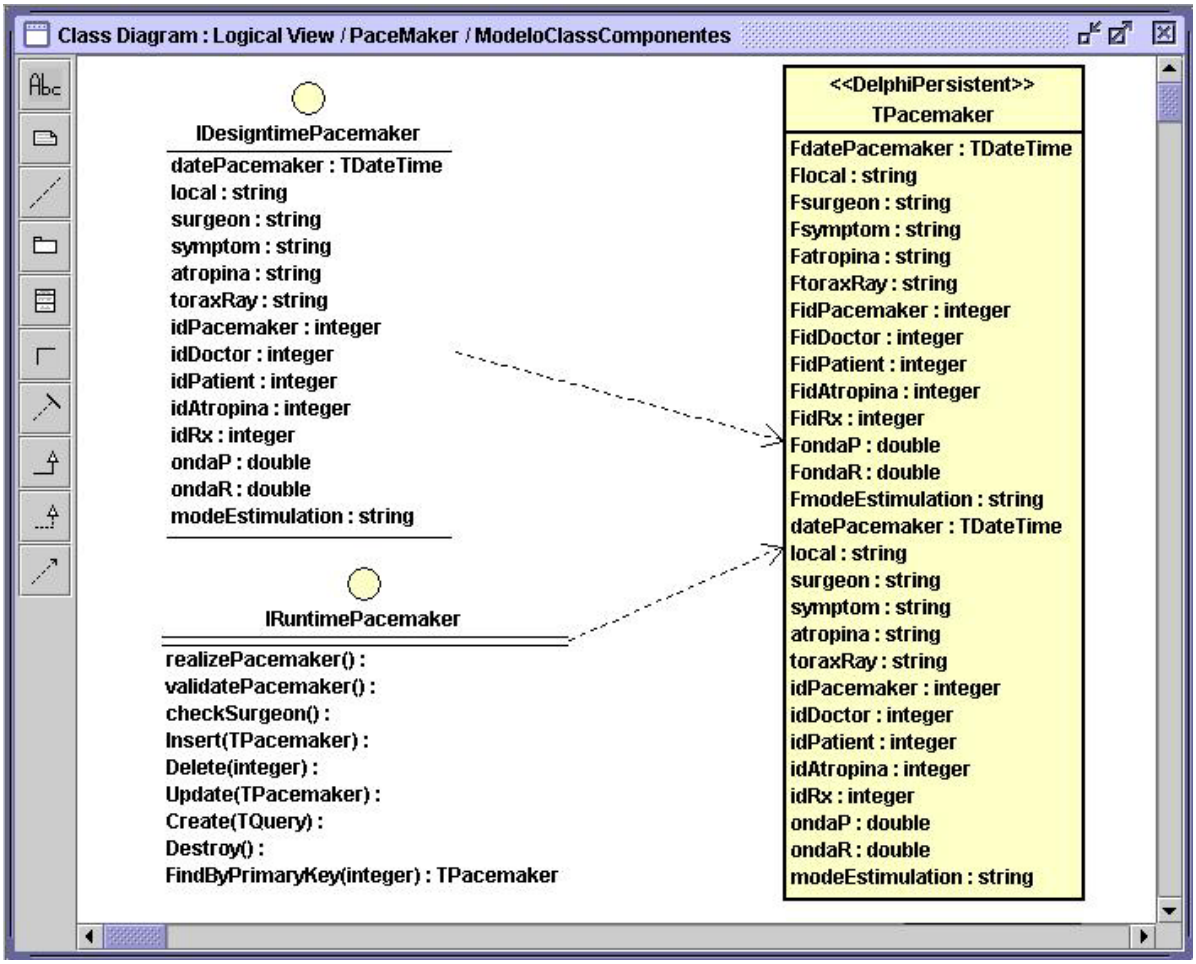


Figura 44– Interfaces do Componente *TPacemaker*

Os métodos de persistência dos componentes do tipo *DelphiPersistent* utilizam a linguagem *SQL* para acesso a banco de dados. Neste passo são também geradas as implementações dos métodos dos componentes na Linguagem *ObjectPascal*, a partir das suas

especificações nos Modelos de Classes. A Figura 45 mostra detalhes do método de localização de instâncias do componente *TPacemaker*, especificado na linguagem *ObjectPascal*.

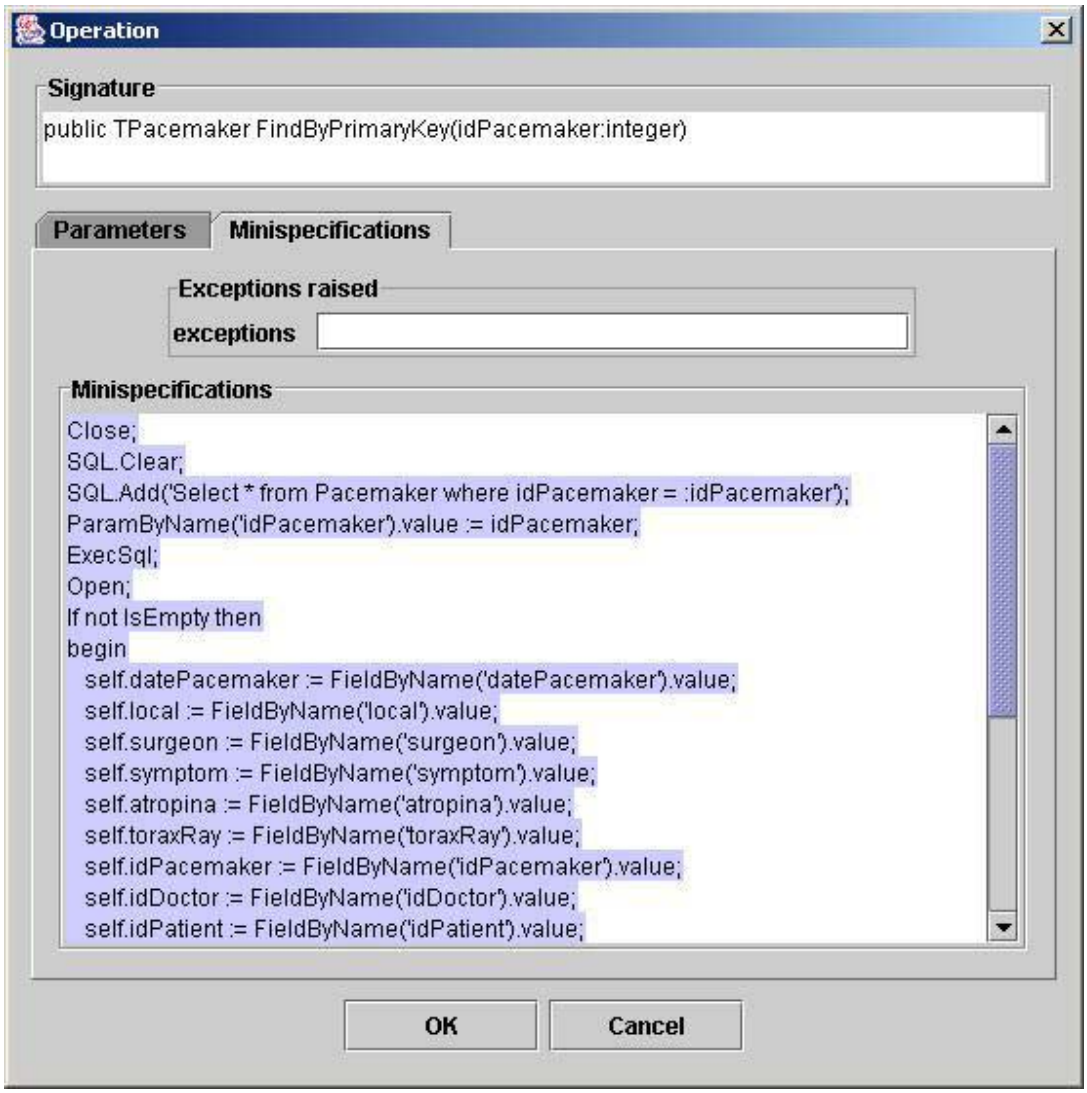


Figura 45– Método de localização de instâncias do componente *TPacemaker*

A Tabela 8 relaciona as principais *stereotypes* usados para organizar os métodos segundo as interfaces *IDesigntime* e *IRuntime* dos componentes. Por exemplo, o *stereotype* <<*DelphiFinder*>> representa os métodos de localização de instâncias dos componentes, disponível na interface *IRuntime*.

Tabela 7 - Métodos Gerados pela ferramenta MVCASE

STEREOTYPE	CATEGORIA DE MÉTODO	MÉTODO GERADO
<<DelphiVirtualFieldGet>>	Acesso aos Atributos	Get<nomeatributo>
<<DelphiVirtualFieldSet>>	Atribuição	Set<nomeatributo>
<<DelphiConstructor>>	Criação de Objetos	Create
<<DelphiDestructor>>	Remoção de Objetos	Destroy
<<DelphiPersistence>>	Persistência	Insert, Update e Delete
<<DelphiFinder>>	Localização de Instâncias	FindByPrimaryKey
<<BusinessRule>>	Regras de Negócios	Especificado pelo Engenheiro Software

4.1.3.3 Projeto do Modelo de Dados

Opcionalmente pode-se gerar os scripts *SQLs* para criação das tabelas do Banco de Dados que fazem a persistência dos componentes do tipo "*Entity*". A partir dos Modelos de Classes são gerados os Modelos de Dados na ferramenta MVCASE, antes da geração do Modelo de Componentes, obtendo-se os scripts *SQL* das classes persistentes. A ferramenta MVCASE suporta a geração automática destes scripts para SGBDs Relacionais, como *MySQL*, *PostgreSQL*, e *Firebird*, usando SQL padrão.

A Figura 46 mostra detalhes da geração do Modelo de Dados a partir do Modelo de Classes, usando a opção "*Transform to Data Model...*" na ferramenta MVCASE.

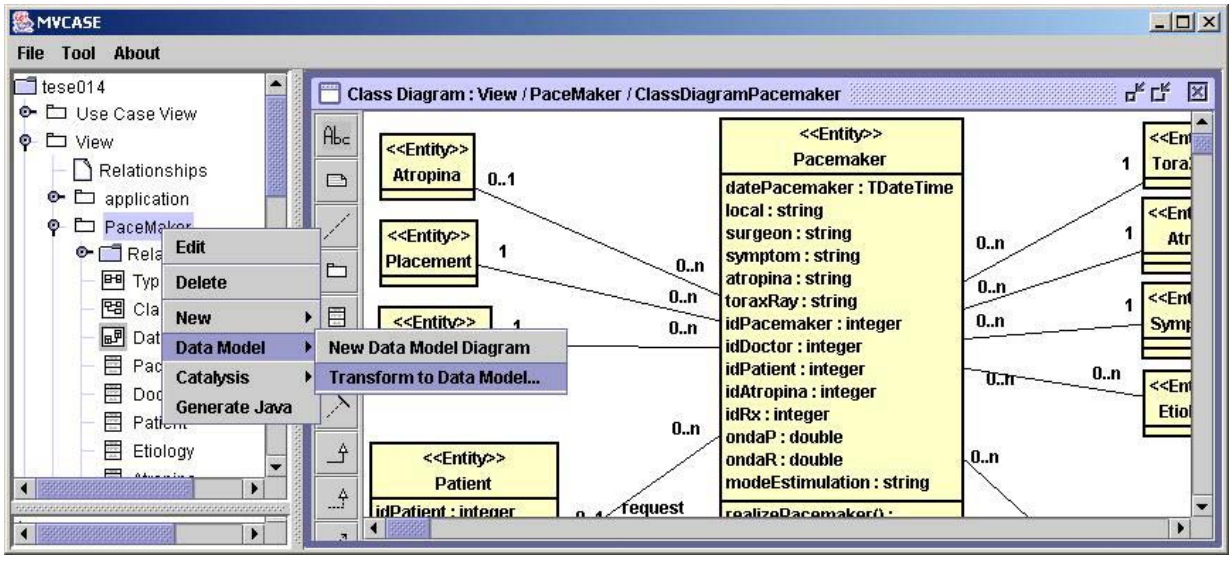


Figura 46– Geração do Modelo de Dados a partir do Modelo de Classes

A Figura 47 mostra o Modelo de Dados gerado pela ferramenta MVCASE, onde existe um relacionamento 1:n entre as tabelas *Patient* e *Pacemaker*.

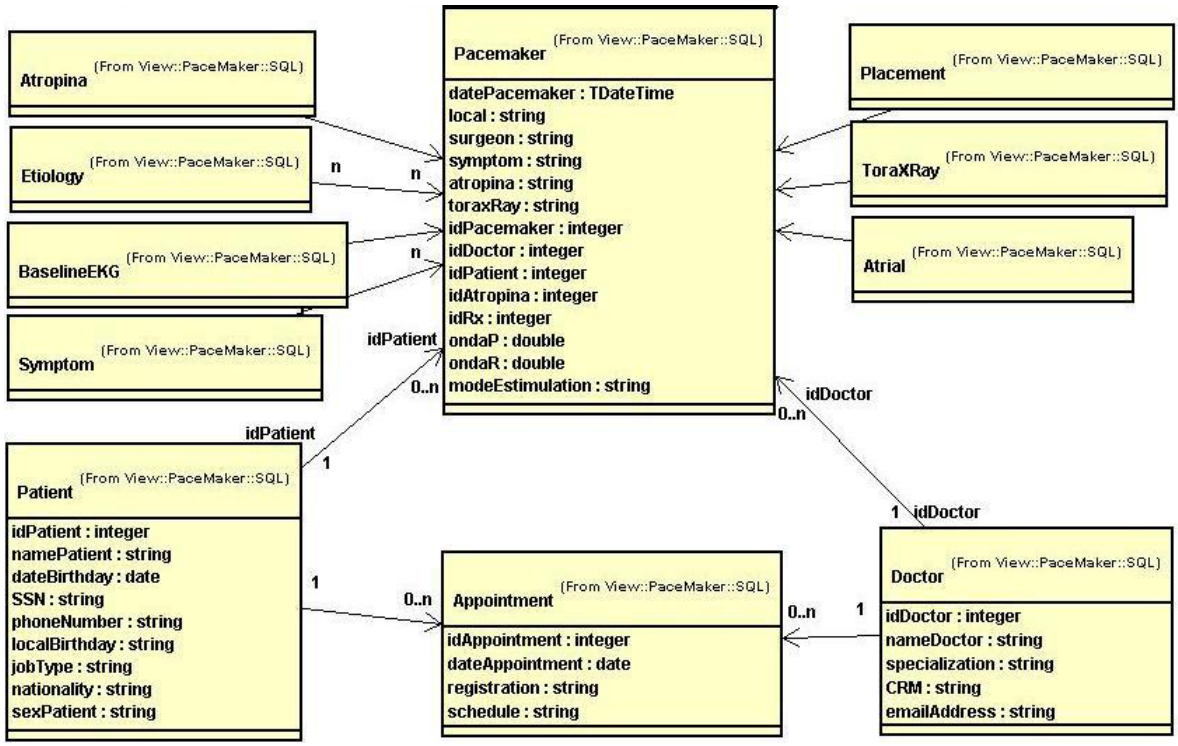


Figura 47– Modelo de Dados obtido a partir do Modelo de Classes

A Figura 48 mostra parte de um script *SQL* gerado, que cria a tabela *Pacemaker,* para persistência do componente *TPacemaker*.

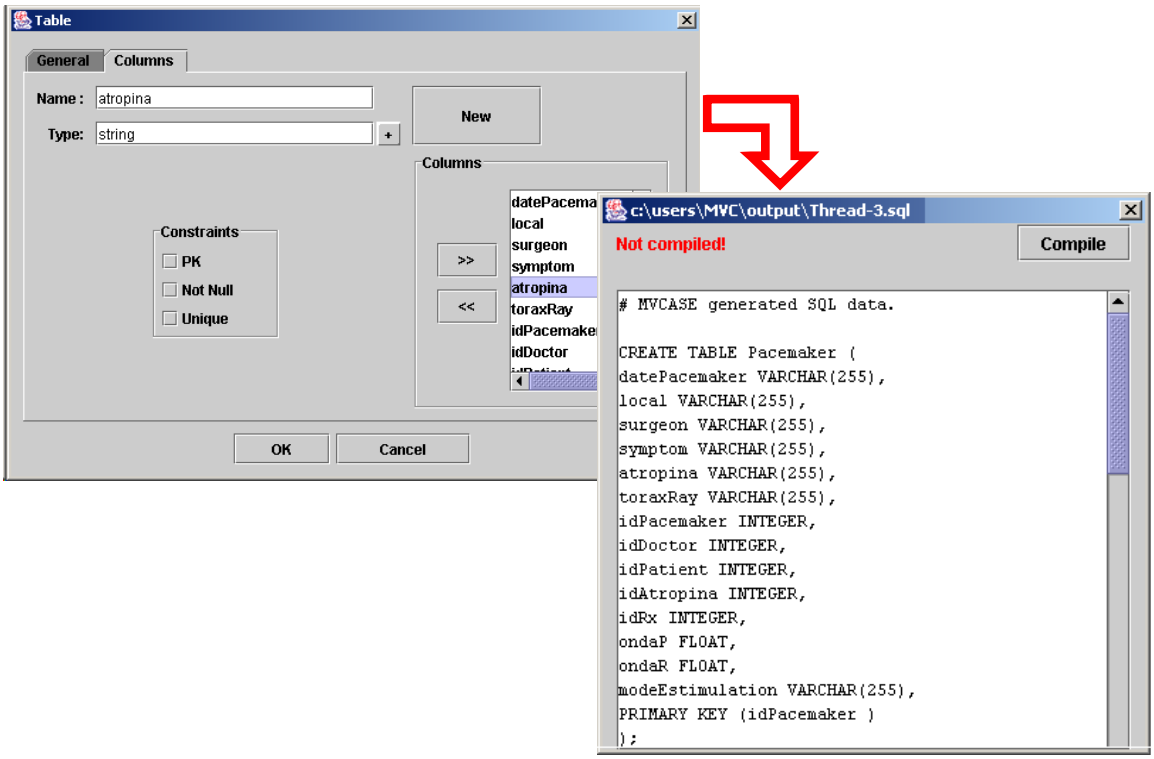


Figura 48– Script *SQL* gerado pela ferramenta MVCASE

Após a geração dos scripts *SQL* o engenheiro de software executa o script para a criação das tabelas que fazem a persistência dos componentes *entity*. A Figura 49 mostra a execução do script *SQL* para o SGBD relacional *Firebird 1.0,* criando a tabela *Pacemaker* para persistência do componente *TPacemaker*. No caso foi utilizada a ferramenta *IBExpert 2.0* para facilitar a execução do script e criação das tabelas.

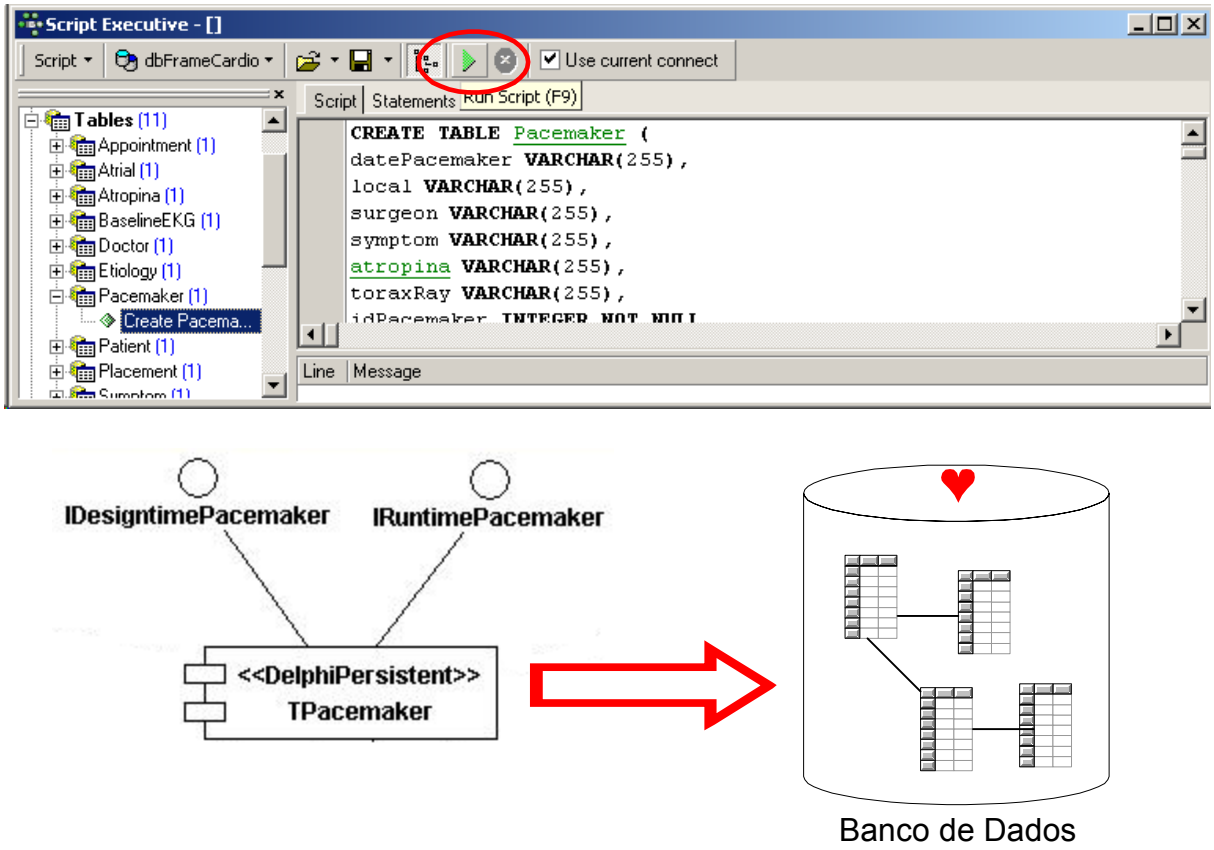


Figura 49– Execução do Script *SQL* para o SGBD relacional *Firebird 1.0*

Em resumo, as principais atividades do passo *Projetar Componentes* incluem:

- Refinamento dos Modelos de Tipos obtendo os Modelos de Classes com seus atributos, métodos, e relacionamentos, levando em consideração a arquitetura dos componentes com suas interfaces e os requisitos não funcionais;
- Refinamento dos Modelos de Interações, representados através dos diagramas de seqüência, para mostrar detalhes de projeto dos comportamentos dos métodos em cada classe; e

- Criação do Modelo de Componentes, a partir dos Modelos de Classes, onde são utilizados os *stereotypes* <<*DelphiTransient*>> e <<*DelphiPersistent*>> para distinguir componentes transientes e persistentes, e outros *stereotypes* que orientam a organização dos métodos e atributos nas classes e interfaces dos componentes. Este modelo representa os componentes com suas interfaces, *IDesigntime e IRuntime* para "*plug-in*" e suas dependências, semelhante à arquitetura VCL do *Delphi*.

4.1.4 Considerações Finais

Esta seção apresentou, de forma detalhada, a modelagem dos componentes em três passos: Definir Domínio do Problema, Especificar Componentes e Projetar Componentes, segundo o método Catalysis.

No passo Definir Domínio do Problema os modelos de *mind-map*, definidos na identificação do problema, são especificados num Modelo de Tipos de Ações, e, posteriormente, particionado e refinado em Modelos de Casos de Uso, visando diminuir a complexidade e melhorar o entendimento do domínio do problema. Uma vez definido o domínio do problema o engenheiro de software passa para o próximo passo da Abordagem, para especificar com mais detalhes os modelos dos componentes.

No passo Especificar Componentes, as atividades realizadas pelo engenheiro de software, na ferramenta MVCASE, compreendem: a especificação do Modelo de Tipos, obtido do refinamento do Modelo de Tipos de Ações e da generalização dos *Snapshots*; a criação dos Modelos de Colaborações entre os objetos envolvidos na realização dos casos de uso; e a criação dos Modelos de Interações, representados pelos diagramas de seqüência, que detalham os comportamentos dos casos de uso.

Os Modelos de Tipos e de Interações, são refinados no passo *Projetar Componentes*, para obter o projeto interno dos componentes. A partir do refinamento dos Modelos de Tipos obtém-se os Modelos de Classes, onde são representadas as classes e seus relacionamentos, levando em consideração a definição dos componentes com suas interfaces. Os Modelos de Interações, representados através dos diagramas de seqüência, são refinados para mostrar detalhes de projeto dos comportamentos dos métodos em cada classe.

A partir dos Modelos de Classes, obtém-se automaticamente uma primeira versão do Modelo de Componentes. São utilizados os *stereotypes* <<*DelphiTransient*>> e

<<*DelphiPersistent*>> para distinguir componentes transientes e persistentes. Este modelo representa os componentes com suas interfaces para conexão e suas dependências, semelhante à arquitetura VCL, onde cada componente disponibiliza seus serviços através de duas interfaces, *Designtime e Runtime*.

Opcionalmente pode-se gerar os scripts *SQLs* para criação das tabelas do Banco de Dados que fazem a persistência dos componentes do tipo "*Entity*". A partir dos Modelos de Classes são gerados os Modelos de Dados na ferramenta MVCASE, obtendo-se os scripts *SQL* das classes persistentes. A ferramenta MVCASE suporta a geração automática destes scripts para SGBDs Relacionais, como *MySQL*, *PostgreSQL*, e *Firebird*.

Desta forma o engenheiro de software termina a modelagem dos componentes, que é o primeiro passo para a construção, e passa a gerar o código dos componentes.

4.2 Gerar Código dos Componentes

No passo *Gerar Código dos Componentes*, com base nos Modelos de Componentes e no Modelo de Classes, obtidos do *Projeto Interno dos Componentes*, obtêm-se as descrições dos componentes na linguagem de modelagem *MDL*, conforme mostra a Figura 50. As descrições *MDL* contêm as especificações dos métodos dos componentes diretamente na Linguagem *ObjectPascal* (*Delphi*).

Neste passo, usando o Sistema de Transformação (ST) Draco-PUC, faz-se a geração do código dos componentes a partir de suas descrições em *MDL*.

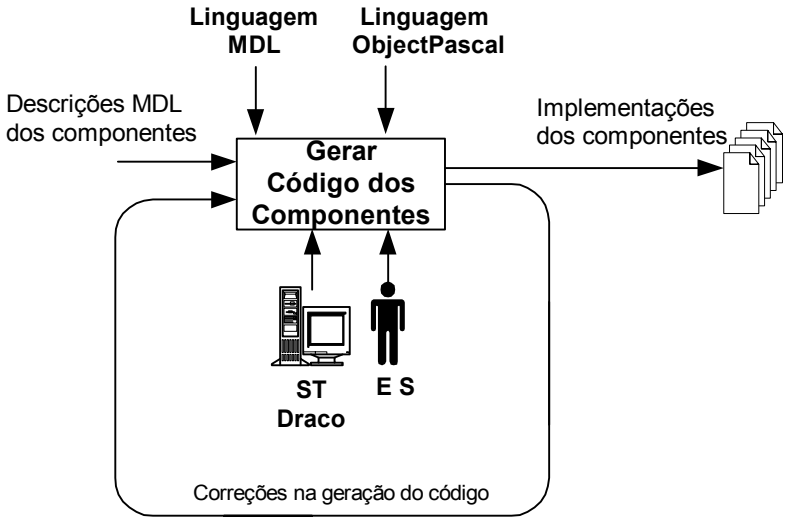


Figura 50–Passo *Gerar Código dos Componentes*

Para gerar o código no Sistema de Transformação Draco-PUC foi necessário construir o domínio *MDL* e o domínio *ObjectPascal*, com seus *parsers* e *pretty-printers*.

O domínio *MDL* já está disponível no Sistema de Transformação Draco-PUC, fruto de outra pesquisa [Fukuda, 2000]. Para o domínio *Delphi* [Moraes, 2001], foi necessária a construção do *parser* da linguagem *ObjectPascal* e o correspondente *prettyprinter*. O domínio *Delphi* foi desenvolvido a partir da versão 5 da gramática *ObjectPascal*, e da sua documentação disponível [Borland, 2000].

Também é necessário construir transformadores que mapeiam as descrições *MDL* e o código *ObjectPascal*. O transformador *MdlToObjectPascal* foi construído com base nas gramáticas *MDL* e *ObjectPascal*. As regras de produção de ambas gramáticas orientaram o Engenheiro de Software na descrição das transformações. A geração do transformador a partir das descrições das transformações fica a cargo do subsistema *tfmgen,* do Sistema de Transformação Draco-PUC, que é orientado pelos *parsers* das linguagens fonte, *MDL*, e de destino, *ObjectPascal*.

Para construir os transformadores o engenheiro de software baseia-se nas gramáticas das linguagens de origem e alvo das transformações. No caso tem-se a linguagem *MDL* como origem e *ObjectPascal* como alvo da implementação. Assim, as transformações foram escritas para fazer o mapeamento entre as regras da gramática *MDL* e as correspondentes regras da gramática *ObjectPascal*. A Figura 51 mostra parte do mapeamento entre as duas gramáticas, onde, por exemplo, a regra "*body*", que reconhece o corpo de um método na linguagem *MDL,* tem seu mapeamento para a regra "*stmt_list*" (*statement list*) na linguagem *ObjectPascal*.

Gramática *MDL*	**Gramática *ObjectPascal***
class_category_RootCategory	package_file
class_Object	unit_file
classAttribute	not_field_definition
type	type
operations	proc_impl func_impl
body	stmt_list
. . .	. . .

Figura **51**– Mapeamento entre as regras das gramáticas *MDL* e *ObjectPascal*

As transformações foram construídas em duas fases. Na primeira fase são lidas as descrições MDL, onde são reconhecidos fatos sobre os pacotes, as classes, os métodos e atributos, e são armazenados na base de conhecimento do ST Draco-PUC *(KB – Knowledge Base)* , e numa segunda fase os fatos são lidos da base de conhecimento passando a gerar o código *ObjectPascal* com base nos modelos de classes e de componentes.

Para que se tenha uma idéia de como atuam as transformações construídas no ST Draco-PUC, a Figura 52 mostra um reconhecimento de uma classe com seus métodos e operações. Do lado esquerdo, em (1) tem-se o padrão de reconhecimento, no ponto de controle *LHS*, da declaração *Logical_Model_Classes_1* em *MDL,* em (2) tem-se o método *KBAssertIfNew* ( ) que armazena o fato "Class" na base de conhecimento. Do lado direito, em (3) têm-se os fatos relacionados a uma *Classe*, armazenados na base de conhecimento.

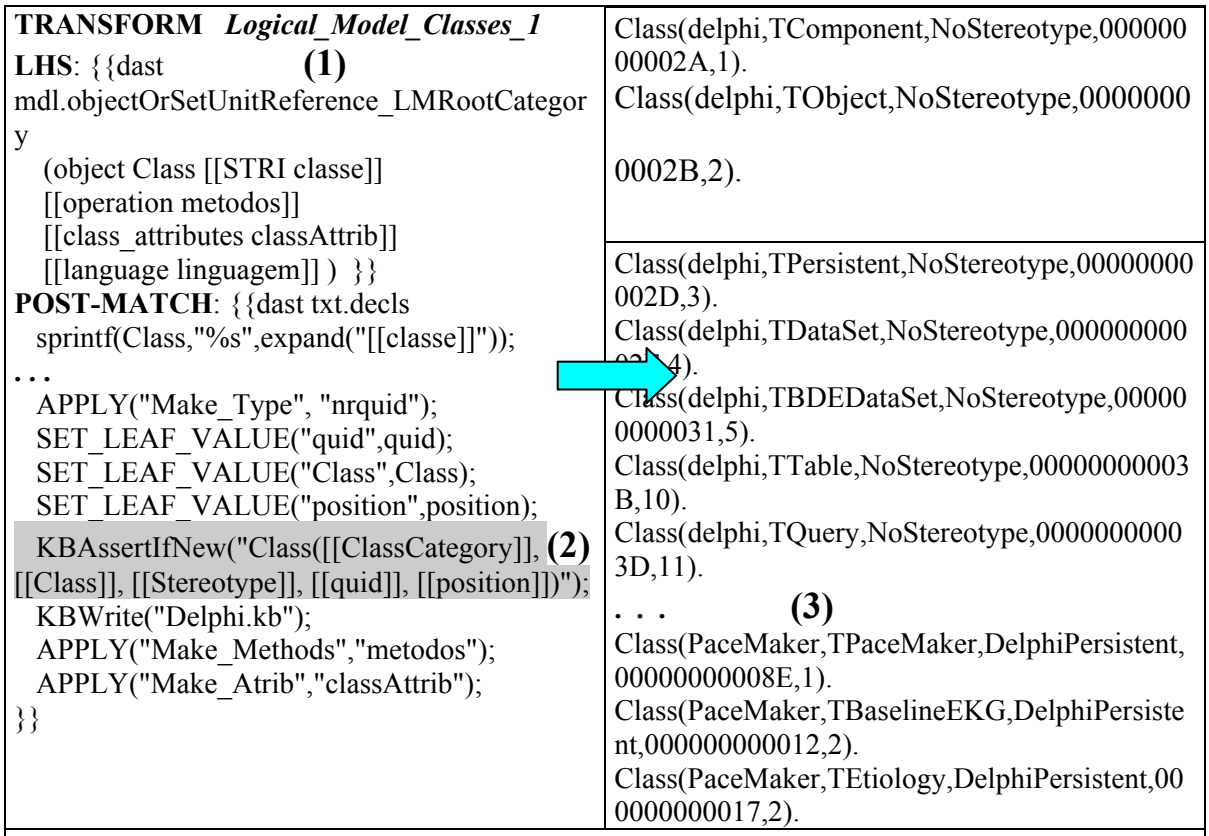


Figura 52- Parte do Transformador *MDLToObjectPascal*

A Figura 53 apresenta outro exemplo de transformação da especificação de um método em MDL para a linguagem de implementação Object Pascal. Do lado esquerdo, em (1) tem-se em destaque o padrão de reconhecimento *Logical_Model_Classes_100*, no ponto de controle *LHS*, que reconhece a declaração e o corpo de um método na linguagem origem, no

caso *MDL*, em (2) tem-se os templates *Semantic* e *BodyProcedure* que armazenam os valores das metavariáveis em *workspaces*. Do lado direito, no ponto de controle *RHS,* em (3) tem-se o correspondente padrão de substituição *Semantic* e *BodyProcedure*, que descreve o corpo do método linguagem *ObjectPascal*.

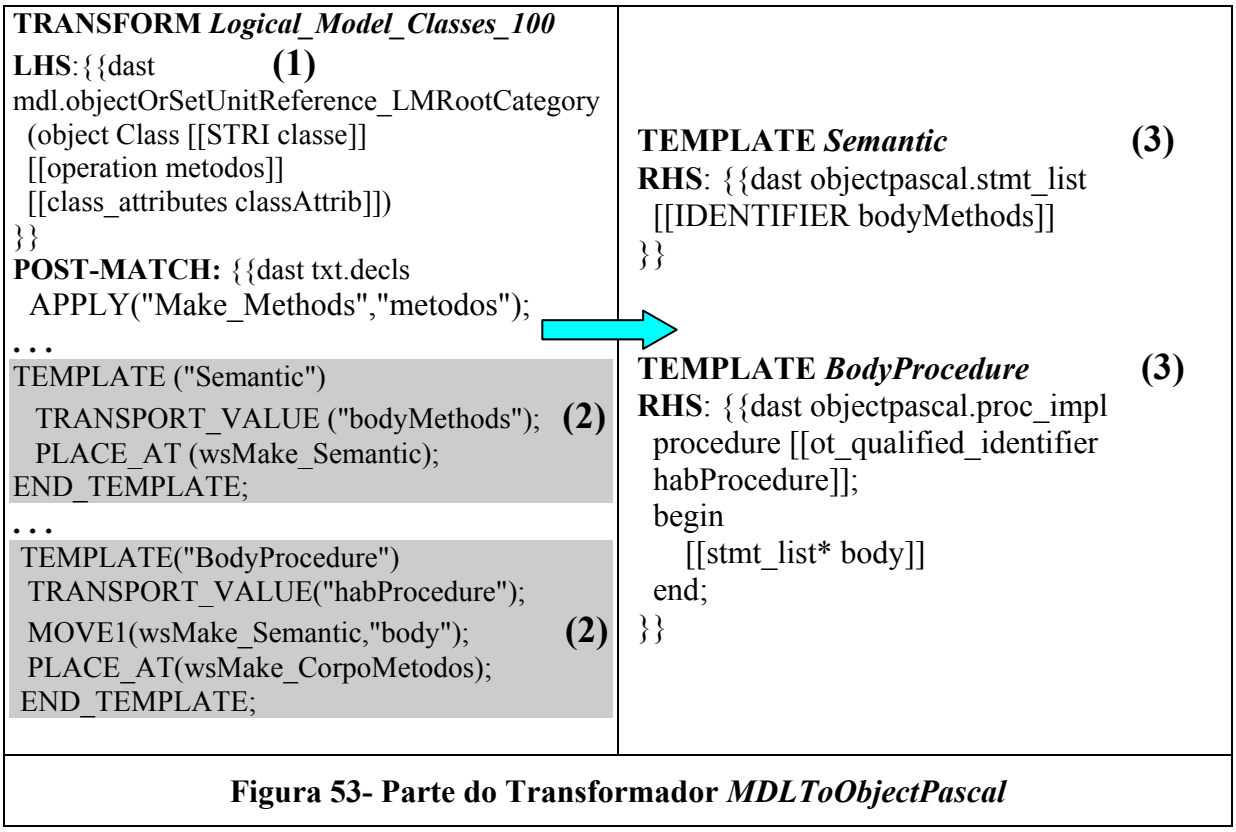


Figura 53- Parte do Transformador *MDLToObjectPascal*

Da mesma forma, foram escritas as demais transformações que mapeiam as descrições em *MDL* para a linguagem alvo *ObjectPascal*.

Para que se tenha uma idéia da geração de código obtida com as transformações, a Figura 54 mostra, à esquerda, a descrição *MDL* de um Modelo de Componentes, e à direita, o correspondente código *ObjectPascal* gerado. As descrições em *MDL*, contêm mini-especificações dos métodos das classes diretamente na linguagem alvo da implementação, no caso o *ObjectPascal*.

Na Figura 54, tem-se em (1) o reconhecimento da classe *TPacemaker* à esquerda, e o nome da *Unit* à direita, *UnitTPacemaker*, e o tipo *TPacemaker*, que é derivado da classe *TQuery*, em (2) tem-se o reconhecimento dos atributos *FdatePacemaker* e *Flocal* na linguagem *MDL*, e à direita os correspondentes atributos em *ObjectPascal*, e em (3) tem-se à esquerda a declaração e o corpo do método *FindByPrimaryKey( )* descrito em *MDL,* e à direita o mesmo código portado pelas transformações, usando o ST Draco-PUC.

Especificação *MDL*

```
(object Class "TPacemaker"    (1)

class_attributes (list class_attribute_list
 (object ClassAttribute "FdatePacemaker" type
"TDatetime")              (2)
 (object ClassAttribute "Flocal" type "string")

. . .

  (object Operation  "FindByPrimaryKey"  (3)
  Close;
  SQL.Clear;
  SQL.Add('Select * from Pacemaker where
       idPacemaker = :idPacemaker');
  ParamByName('idPacemaker').value :=
       idPacemaker;
  ExecSql;
  Open;
  If not IsEmpty then
  begin
     self.datePacemaker :=
     FieldByName('datePacemaker').value;
     self.local := FieldByName('local').value;
. . .
  begin
     result := nil;
  end; )
```

Código *ObjectPascal* Gerado

```
unit UnitTPacemaker;    (1)
interface
uses Windows, Messages, SysUtils, Classes,
Dialogs, DB, DBTables;
Type
  TPacemaker = class(TQuery)    (1)
       FdatePacemaker: TDateTime;    (2)
       Flocal:string;
. . .
function TPacemaker.FindByPrimaryKey ( idPacemaker:integer):
TPacemaker;    (3)
  Close;
  SQL.Clear;
  SQL.Add('Select * from Pacemaker where
       idPacemaker = :idPacemaker');
  ParamByName('idPacemaker').value :=
       idPacemaker;
  ExecSql;
  Open;
  If not IsEmpty then
  begin
     self.datePacemaker :=
     FieldByName('datePacemaker').value;
     self.local := FieldByName('local').value;
. . .
  begin
     result := nil;
  end; )
```

Figura 54 – Geração de Código *ObjectPascal*, a partir de especificações MDL.

Após as transformações têm-se os componentes implementados em *ObjectPascal*. Para facilitar o reuso, os componentes são organizados em pacotes, gerando um arquivo com extensão *.dpk* (*dephi package* – pacote de componentes), que contêm as *Units*, com extensão *.pas (pascal source* - código fonte*)*, geradas para cada componente.

A Figura 55 mostra parte do código do arquivo *Pacemaker.dpk* (1), gerado após as transformações, contendo as *Unit*s (2) que faz a implementação dos componentes contidos no pacote *Pacemaker*.

```
package Pacemaker;   (1)
{$R *.res}
{$DESCRIPTION 'Pacemaker'}

requires  rtl, vcl, dbrtl, bdertl;

contains
  UnitTPacemaker in 'UnitTPacemaker.pas',
  UnitTBaselineEKG in 'UnitTBaselineEKG.pas',
  UnitTEtiology in 'UnitTEtiology.pas',                    (2)
  UnitTAtropina in 'UnitTAtropina.pas',
  UnitTThoraXray in 'UnitTThoraXray.pas',
  UnitTSymptom in 'UnitTSymptom.pas',
  UnitTAtrial in 'UnitTAtrial.pas',
. . .;
end.
```

Figura 55**– Pacote gerado pelo ST Draco-PUC**

Depois de gerado o código dos componentes, o engenheiro de software instala os componentes implementados para reuso nas aplicações no ambiente *Delphi*.

4.2.1 Considerações Finais

Esta seção apresentou, de forma detalhada, a geração de código dos componentes, onde os códigos são gerados com base nos Modelos de Componentes e no Modelo de Classes, obtidos do *Projeto Interno dos Componentes*. As descrições *MDL* obtidas do projeto interno dos componentes, contêm as especificações dos métodos dos componentes diretamente na Linguagem *ObjectPascal* (*Delphi*). O ST Draco-PUC é o principal mecanismo utilizado neste passo.

Foi construído o transformador *MdlToObjectPascal* com base nas gramáticas *MDL* e *ObjectPascal,* no Sistema de Transformação Draco-PUC. As transformações foram escritas para fazer o mapeamento entre as regras da gramática *MDL* e as correspondentes regras da gramática *ObjectPascal*.

Deste modo o engenheiro de software conclui o segundo passo para construção dos componentes, obtendo o código dos componentes do domínio de cardiologia na linguagem *ObjectPascal*, e passa a instalar os componentes no ambiente *Delphi*.

4.3 Instalar Componentes

A Figura 56 mostra que a partir do código dos componentes, orientado pela linguagem *ObjectPascal*, obtêm-se os componentes instalados no ambiente *Delphi*.

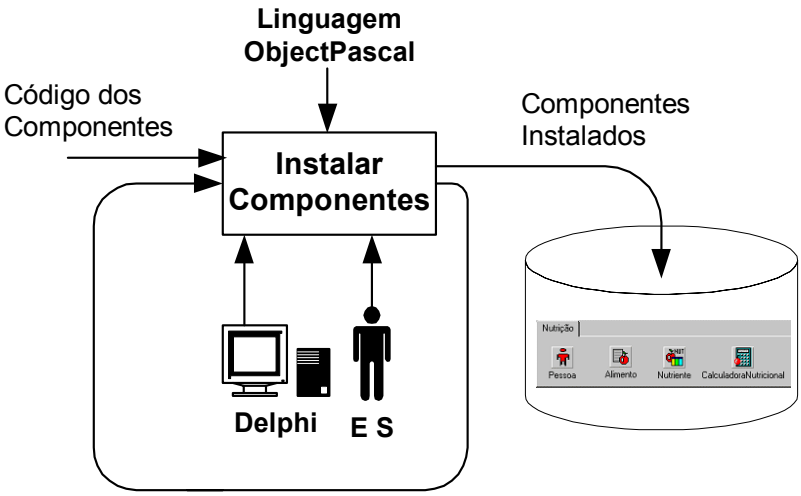


Figura 56– Passo *Instalar Componentes*.

Para instalar os componentes é necessário compilar os componentes organizados em pacotes. No ambiente Delphi o engenheiro de software irá abrir cada um dos pacotes, gerados pelo passo anterior, através da opção *Component → Install VCL Component...*.

A Figura 57 mostra o pacote *Pacemaker.dpk* selecionado para ser instalado. A versão 8 do Delphi, *Borland Delphi 8.0 for the Microsoft .NET Framework*, o pacote selecionado é transformado em *Pacemaker.dll*. A instalação foi testada também nas versões 6 e 7.

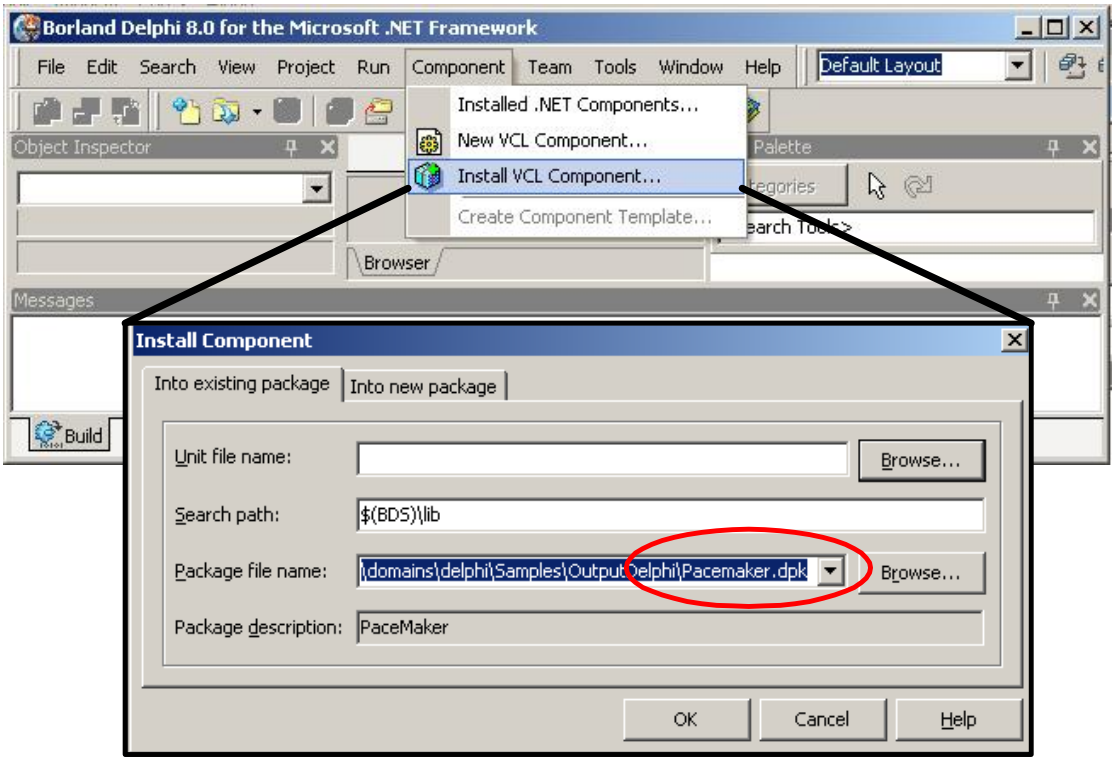


Figura 57– Instalação dos Componentes no ambiente *Delphi*.

Depois de selecionado o pacote a ser instalado, o engenheiro de software irá compilar o pacote contendo as *units*. A Figura 58 mostra a compilação do pacote *Pacemaker*.

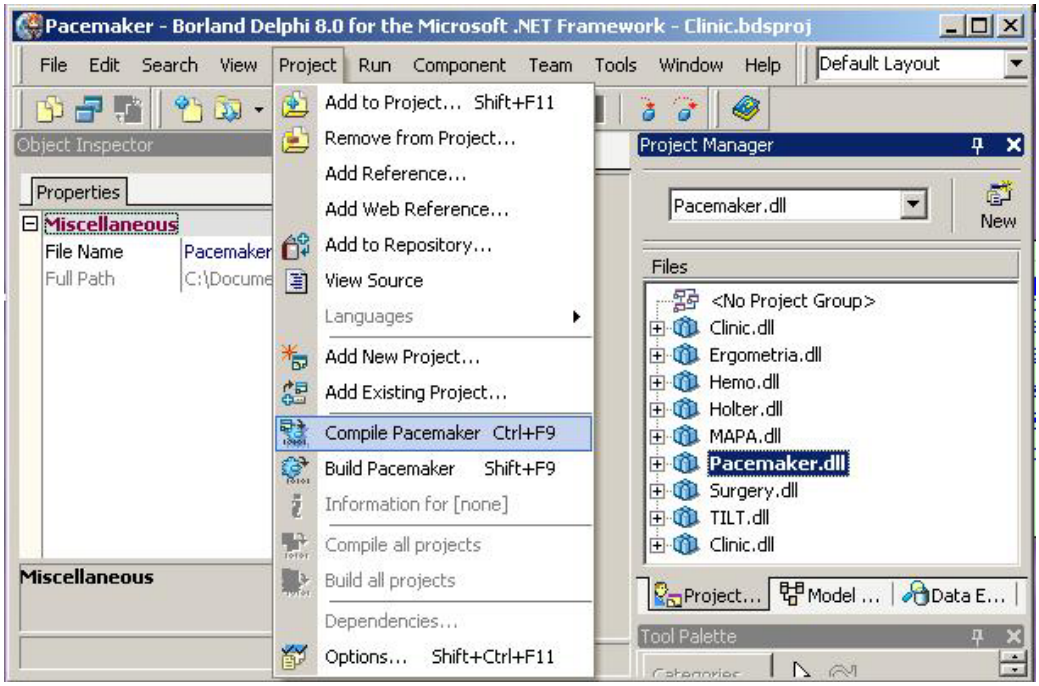


Figura 58– Compilação dos Componentes no ambiente *Delphi*.

Após a compilação dos pacotes, o engenheiro de software irá adicionar os componentes às paletas do Delphi, disponibilizando-os para reuso. A Figura 59 mostra a adição dos componentes no ambiente Delphi, através da opção *Add...*, onde o pacote *Pacemaker* contém as *Units* com a implementação em *ObjectPascal* de cada um dos seus componentes.

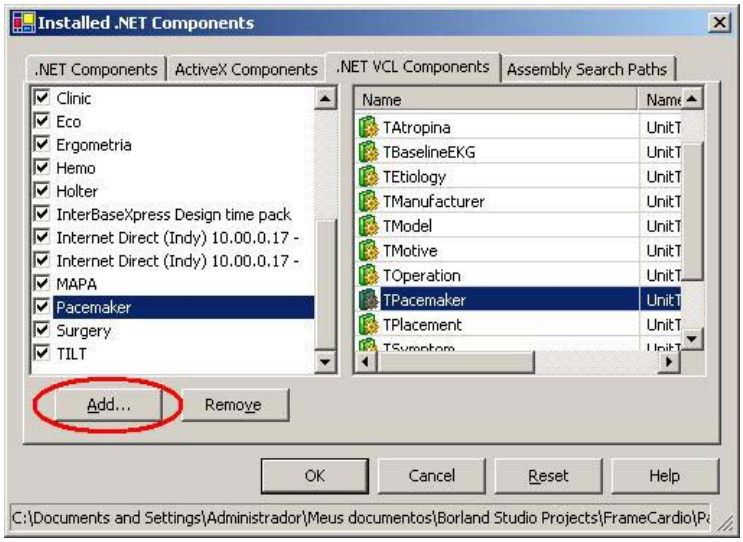


Figura 59– Adição dos Componentes nas Paletas do ambiente *Delphi*.

A Figura 60 mostra parte do código do componente *TPacemaker,* implementado pela classe *TPacemaker,* gerado no passo anterior pelo ST Draco-PUC.

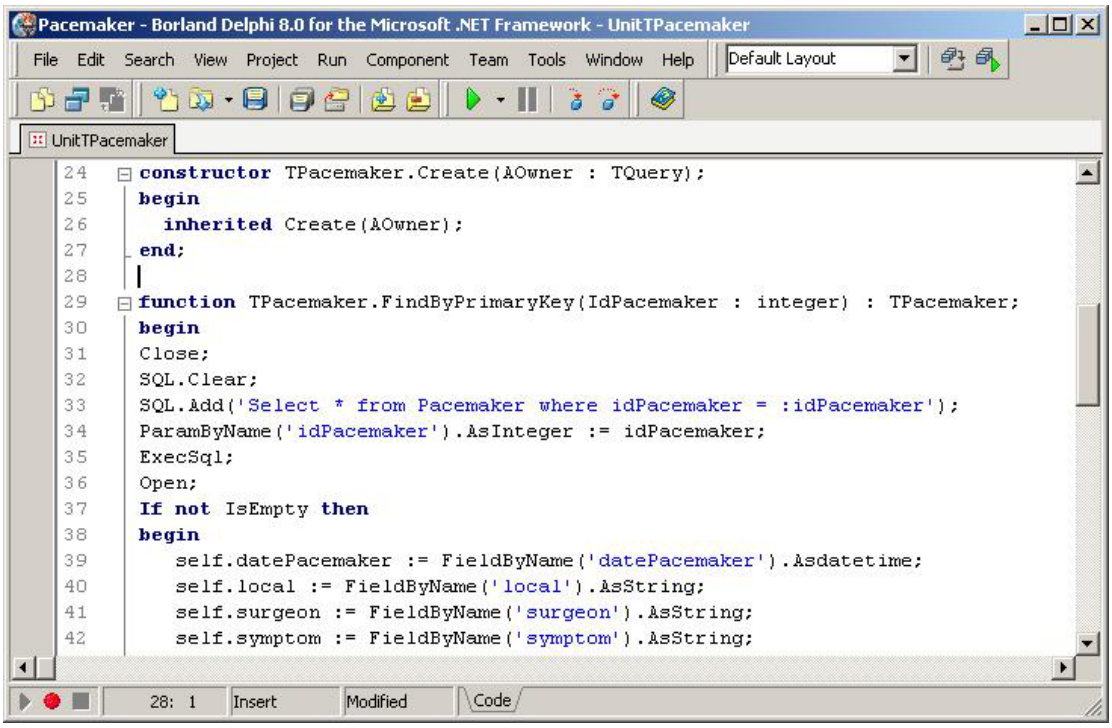


Figura 60– Código do Componente *Pacemaker* no ambiente *Delphi*.

Uma vez instalados, os componentes estarão disponíveis para as aplicações nas paletas no *Delphi*, conforme mostra a Figura 61.

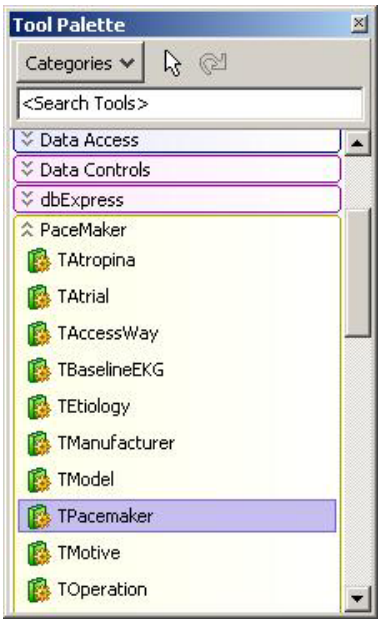


Figura 61– Componentes do Pacote *Pacemaker* no ambiente *Delphi*.

4.3.1 Considerações Finais

Esta seção apresentou, de forma detalhada, a instalação dos componentes no ambiente Delphi. Para instalar os componentes é necessário compilar os componentes organizados em pacotes. Depois de instalados os componentes ficam disponíveis na paleta do Delphi para reuso das aplicações do domínio de cardiologia.

A instalação corresponde ao último passo da construção dos componentes. Sabe-se, contudo que software é um artefato evolutivo e que precisa ser atualizado constantemente. A próxima seção descreve o Ciclo de Vida dos Componentes.

4.4 Ciclo de Vida dos Componentes

O ciclo de vida dos componentes construídos conforme a abordagem proposta, segue as características do modelo Espiral de desenvolvimento de software, no qual o Engenheiro de Software pode retornar aos passos anteriores para refinar os modelos especificados e obter novo projeto dos componentes. A cada ciclo de execução dos passos da abordagem, têm-se novos protótipos dos componentes. Os protótipos permitem verificar se os requisitos especificados estão sendo atendidos, orientando as correções e melhorias dos componentes. Aplicações, reutilizando os componentes, são desenvolvidas para testar os componentes. Foram realizados testes para a depuração e melhoria dos componentes.

Capítulo 5

Reutilização de Componentes

Para melhor compreensão da etapa de reutilização dos componentes, segue-se a apresentação de cada um dos passos do desenvolvimento de uma aplicação que Registra o Implante de Marcapasso em um Paciente, fazendo reuso dos componentes do domínio de cardiologia. Esta aplicação foi desenvolvida para o Centro de Cirurgia Cardíaca de Marília – CCCM.

Para maior clareza e facilitar o entendimento da reutilização dos componentes de um domínio do problema, segue uma apresentação mais detalhada dos passos: *Modelar Aplicação, Gerar Código da Aplicação, Executar Aplicação e Refinar Aplicação*.

5.1 Modelar Aplicação

Neste passo, na ferramenta MVCASE, o engenheiro de software, inicialmente, especifica a aplicação a partir dos seus requisitos. Em seguida, ainda na ferramenta MVCASE, o engenheiro de software faz o projeto da aplicação. Conforme mostra a Figura 62, tanto na especificação como no projeto, reutilizam-se os modelos dos componentes de etapa da construção.

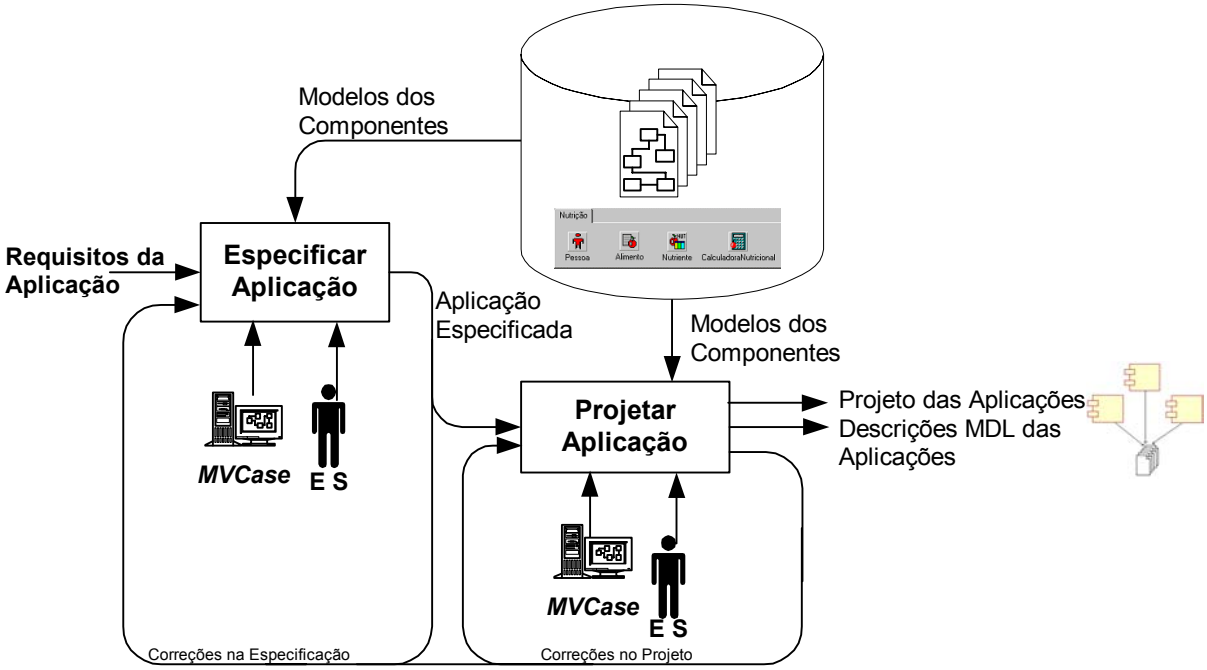


Figura 62– Passo *Modelar Aplicação*

Do mesmo modo que na construção dos componentes, a ferramenta MVCASE é o principal mecanismo para auxiliar o engenheiro de software durante a modelagem da aplicação.

O passo *Especificar Aplicação* tem início com o entendimento do problema, identificando-se os requisitos da aplicação.

> O Sistema do Centro de Cirurgia Cardíaca de Marília tem por finalidade registrar os implantes de marcapasso nos pacientes atendidos na clínica.
>
> Inicialmente o paciente é registrado no sistema antes de ser submetido ao implante. Com o paciente cadastrado, o médico inicia uma investigação diagnóstica sobre o paciente.
>
> Em seguida o médico realiza o implante do marcapasso no paciente, registrando o modo de estimulação utilizado no implante, o tipo de gerador implantado, o motivo do implante do marcapasso no paciente e local onde o marcapasso foi instalado.

Antes de iniciar a especificação na ferramenta MVCASE, o engenheiro de software importa os Modelos dos Componentes do domínio do problema, no caso de Cardiologia, no *browser* da ferramenta MVCASE. A Figura 63 mostra a importação dos componentes na ferramenta MVCASE.

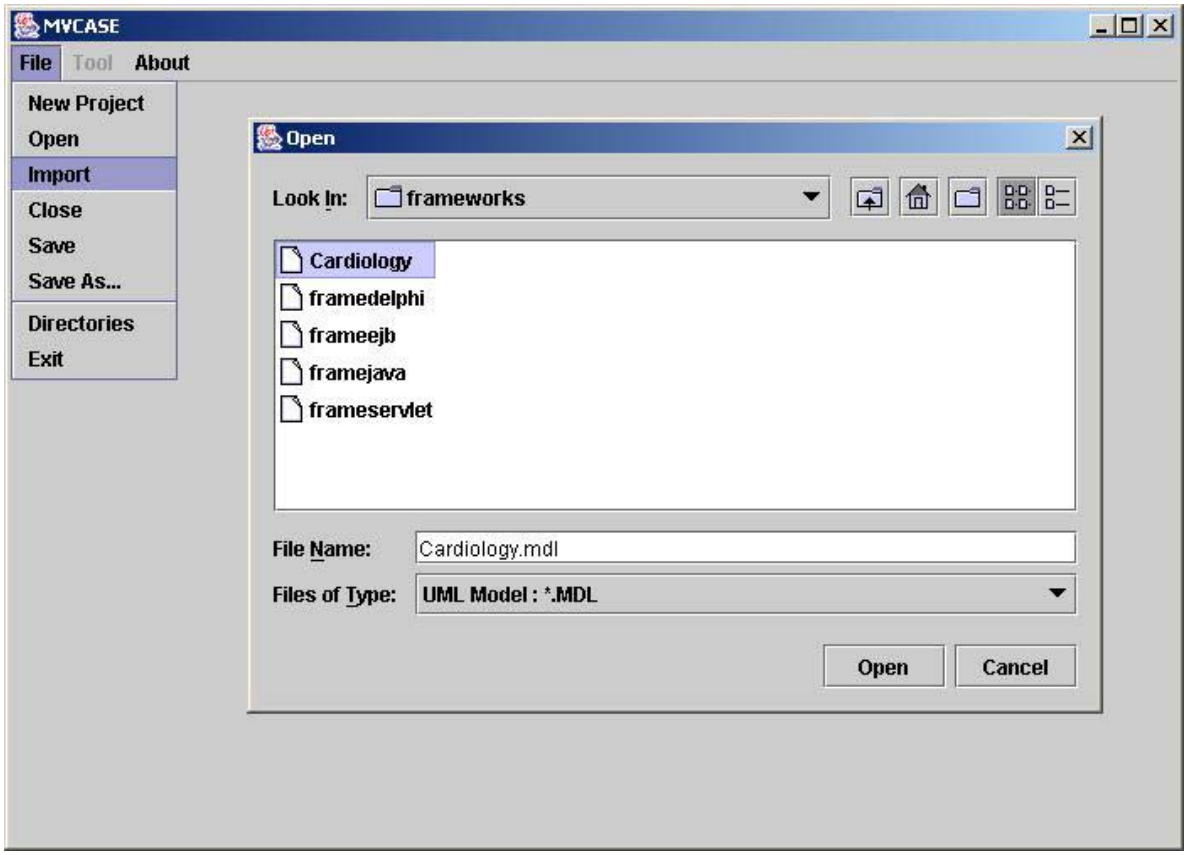


Figura 63– Importação dos Componentes Construídos

Após a importação dos componentes do domínio de cardiologia no *browser* da ferramenta MVCASE, o engenheiro de software deve criar um novo pacote denominado *application*, e definir seu *stereotype* como *<<Delphi Application>>*. A aplicação será desenvolvida dentro deste pacote criado.

Na Figura 64 pode-se observar, à esquerda, os pacotes do domínio de cardiologia e do *delphi*, importados no *browser* da ferramenta MVCASE. A Figura 64 mostra também a organização dos pacotes e suas dependências. Em (1) tem-se o pacote *delphi,* com *stereotype <<Delphi Framework>>*, que contém os componentes da arquitetura *VCL*, em (2) tem-se o pacote *application <<Delphi Application>>* onde serão desenvolvidas as aplicações, e em (3) têm-se alguns dos pacotes do domínio de cardiologia, com *stereotype <<Cardiology>>*, que serão reutilizados pelas aplicações.

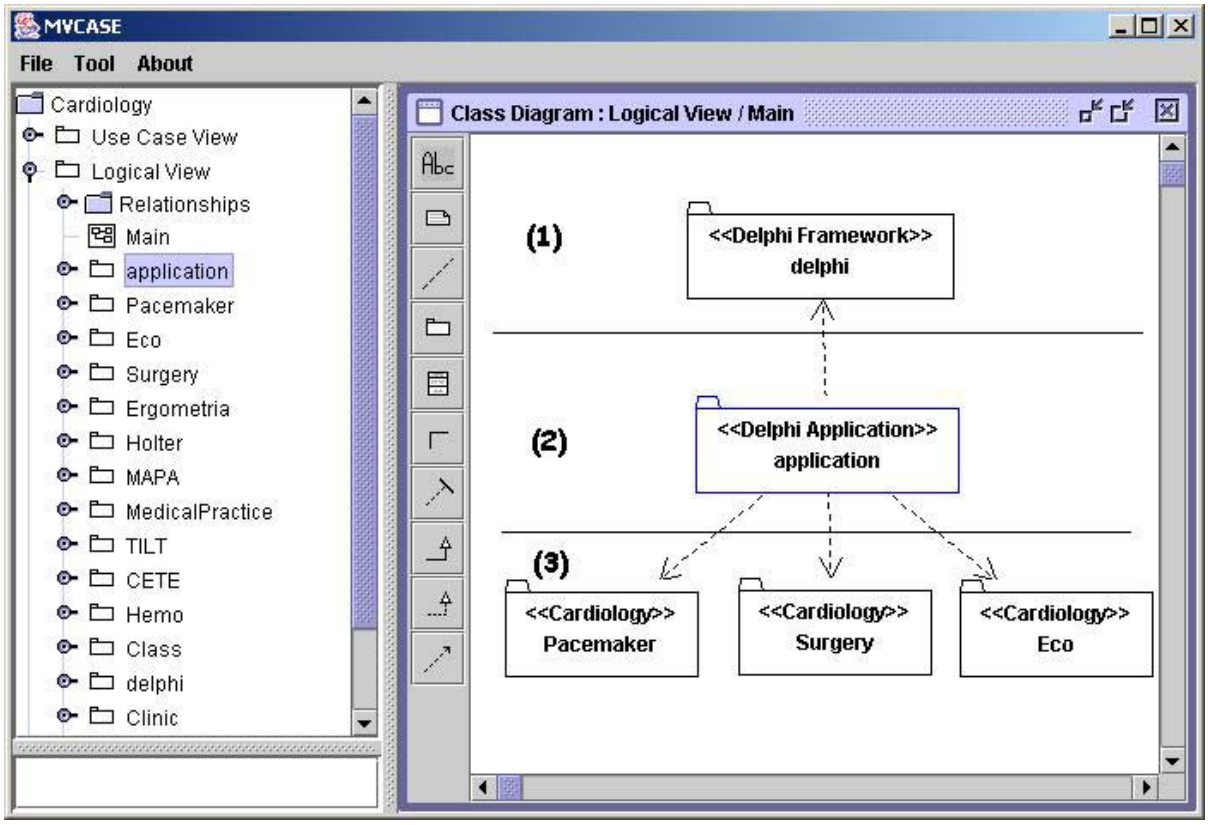


Figura 64. Modelo de Pacotes da Aplicação

As principais técnicas UML usadas na especificação dos requisitos são os Diagramas de Casos de Uso e de Seqüência, os Modelos de Colaborações, e os Modelos de Classes. A Figura 65 mostra do Modelo de Classes da Aplicação com suas dependências e associações. Em (1) têm-se as classes da arquitetura VCL da linguagem *ObjectPascal,* dentro do pacote *delphi*. Em (2) têm-se as classes da aplicação que descendem da VCL, dentro do pacote *application*. Em (3) têm-se as classes reutilizadas do domínio de cardiologia. Da mesma forma que ocorre na construção dos componentes, na modelagem da aplicação, o engenheiro de software define os componentes transientes e persistentes através dos *stereotypes* *<<DelphiTransient>>* e *<<DelphiPersistent>>*. Os atributos *Patient1*, *Pacemaker1*, *Doctor1*, e *Atropina1*, representam os objetos da classe *TDMPacemaker*.

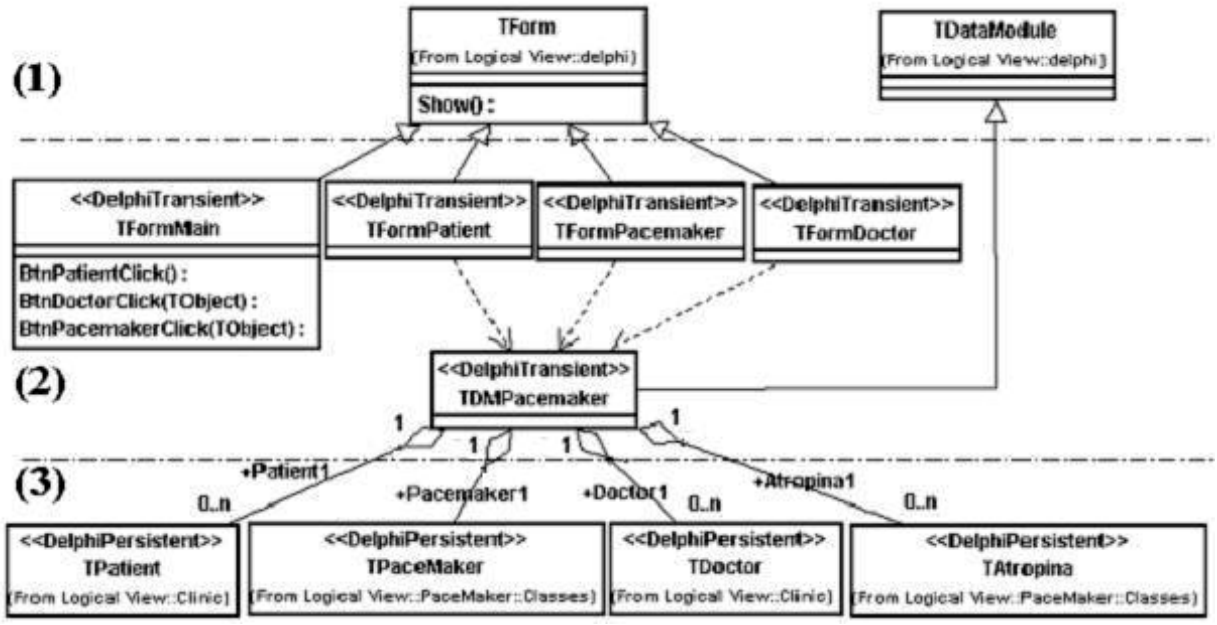


Figura 65. Modelo de Classes da Aplicação

No passo *Projetar Aplicação,* as especificações são refinadas pelo engenheiro de software, para obter o Projeto da Aplicação. Neste passo, são tratados os requisitos não funcionais, relacionados com a arquitetura, distribuição, tratamento de erros, interfaces, persistência dos dados, e outros.

Um dos modelos resultante deste passo é o Modelo de Componentes. Baseado na arquitetura da VCL são construídos os componentes com suas interfaces *IRuntime* e *IDesigntime*. Por exemplo, na Figura 66, têm-se os componentes da arquitetura VCL, da aplicação e do domínio de cardiologia, os quais disponibilizam seus métodos, propriedades e eventos, facilitando a reutilização.

Em (1) têm-se o componente TForm e TDataModule da arquitetura da VCL que são reutilizados pelos componentes da aplicação (2). Em (2) têm-se as dependências entre os componentes da aplicação, onde o componente TFormPacemaker depende do componente TDMPacemaker. Em (3) têm-se os componentes do domínio de cardiologia: TPatient, TPacemaker, TDoctor, e TAtropina, que são reutilizados pelo componente TDMPacemaker da aplicação.

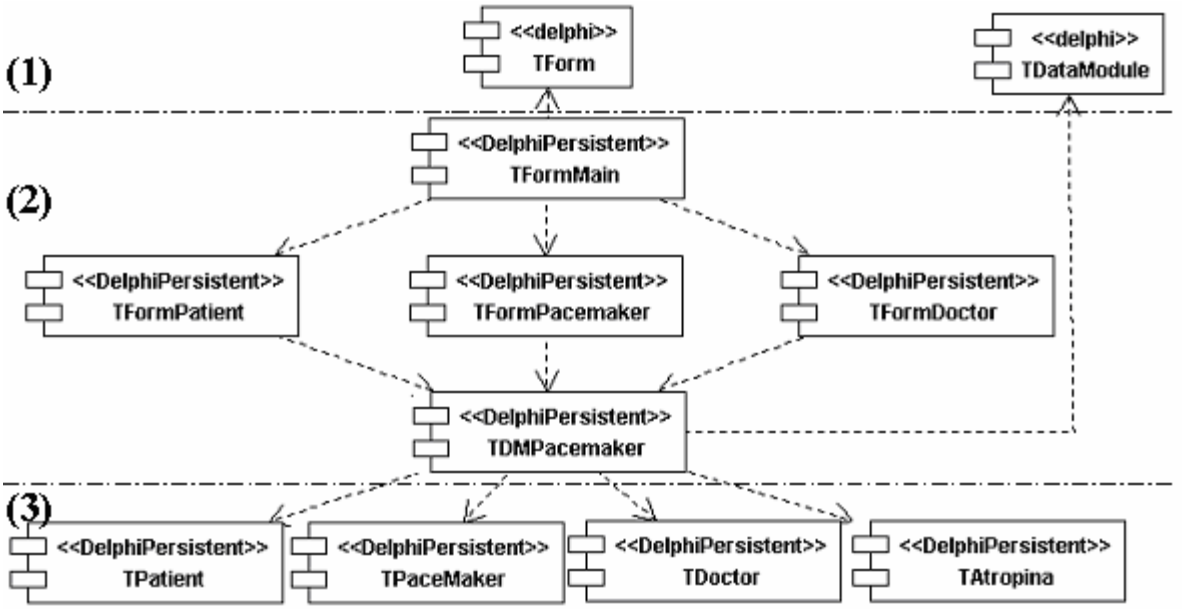


Figura 66. Modelo de Componentes da Aplicação

O engenheiro de software, ainda na ferramenta MVCASE, pode implementar os métodos dos componentes da aplicação. Por exemplo a Figura 67 mostra o protótipo e a mini-especificação do método *FindPatient*( ), da aplicação, que chama o método *FindByPrimaryKey* ( ) do componente TPatient, que retorna uma instância do tipo da classe.

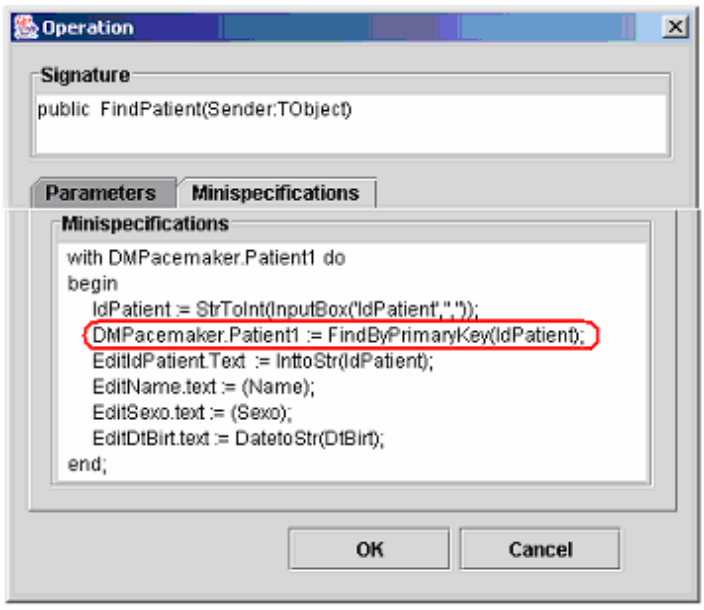


Figura 67. Corpo do Método *FindPatient* da Aplicação

Concluindo o Projeto, o engenheiro de software passa a *gerar o código da aplicação*.

5.2 Gerar Código da Aplicação

A ferramenta MVCASE gera as descrições *MDL*, baseado no Projeto das Aplicações. A partir das descrições *MDL*, utiliza-se o ST Draco-PUC para g*eração do código da aplicação*. O ST Draco faz o mapeamento das descrições *MDL* para a Linguagem *ObjectPascal,* através de transformações, gerando arquivos com extensões .pas (código da aplicação em *units*), .dfm (especificações do formulário) e .dpr (organização das *units*).

A Figura 68 mostra parte do código da aplicação gerado automaticamente pelo ST Draco-PUC. No código estão destacados: em (1) tem-se a declaração da classe TFormPatient, que herda da classe TForm da arquitetura VCL do Delphi, com seus atributos e o método *FindPatient* (2). Em (3) tem-se a importação da UnitDMPacemaker na seção *uses*. Finalmente em (4) tem-se o corpo do método *FindPatient* ( ), destacando-se a chamada do método *FindByPrimaryKey* ( ) do componente TPatient, retornando uma instância da classe TPatient do domínio de cardiologia.

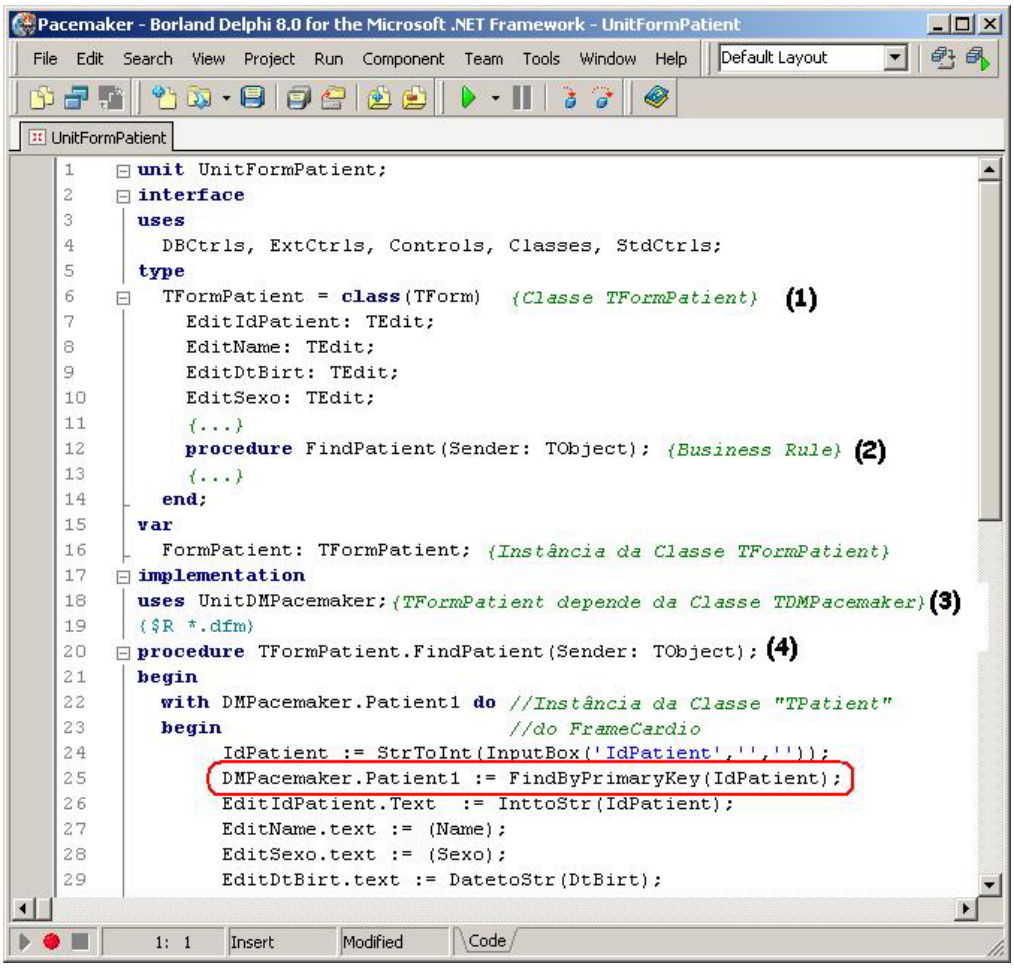


Figura 68– Código Gerado da Aplicação

5.3 Executar Aplicação

No passo *Executar Aplicação* o engenheiro de software utiliza o ambiente *Delphi* para executar e testar o código gerado da aplicação. Conforme mostra a Figura 69, os dados de testes servem para verificar se os requisitos especificados para a aplicação foram atendidos.

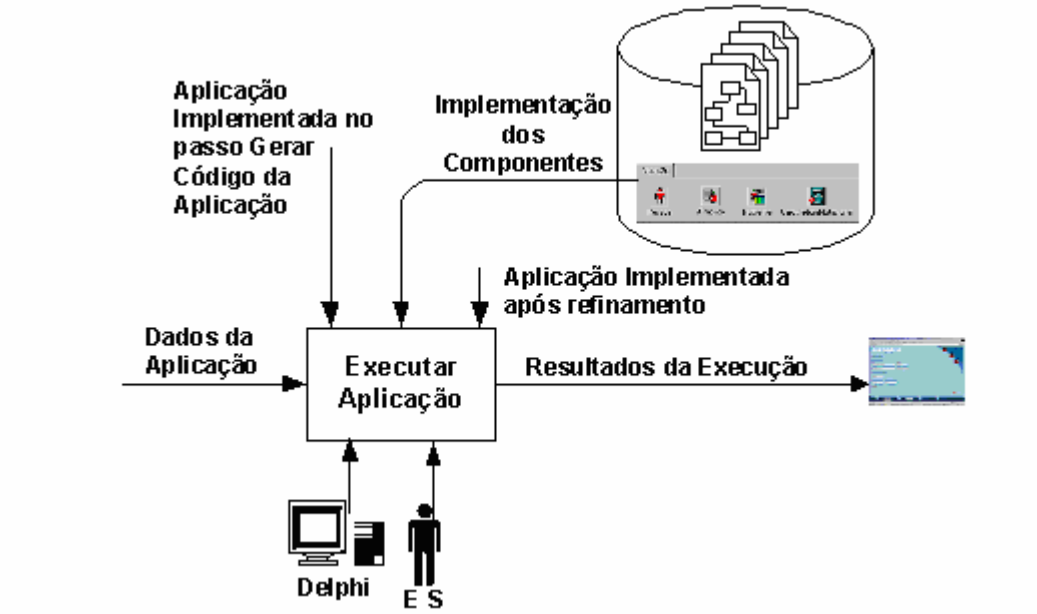


Figura 69– Passo *Executar Aplicação*

A implementação na Linguagem *ObjectPascal*, gerada no passo *Gerar Código da Aplicação* pode não ser suficiente para atender todos os requisitos, principalmente os não funcionais, relacionados com a interface, segurança, tratamento de erros, validação de dados, acesso a banco de dados e outros. Assim, o engenheiro de software, utilizando os recursos do *Delphi*, pode complementar o projeto com outros componentes que implementam estes requisitos. O código gerado pelo Sistema de Transformação Draco-PUC, integrado ao código desenvolvido no *Delphi*, resulta na implementação final da aplicação.

A Figura 70 mostra parte do Diagrama de Componentes visualizado no ambiente *Delphi*, com os componentes *DMPacemaker, Patient1, Doctor1 e Pacemaker1,* reutilizados do domínio de cardiologia. À Esquerda tem-se no *Object Inspector* a interface *Designtime* do componente *Pacemaker1*, disponibilizando suas propriedades, métodos e eventos, para serem

reutilizados na aplicação. Destaca-se ainda no *Object Inspector* a propriedade *Patient* do componente *PaceMaker1* que permite o "*Plug-In*" do componente *Patient1*.

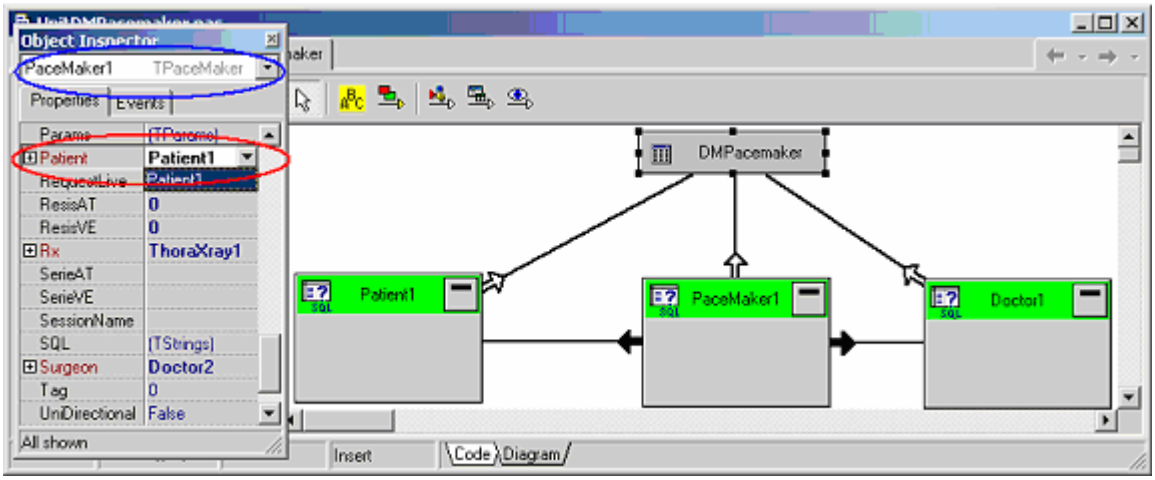


Figura 70. Diagrama de Componentes da Aplicação – IDE Delphi

Estando todo o sistema implementado, pode-se finalmente executá-lo para verificar se o mesmo atende aos requisitos especificados. Caso ocorram problemas, ou surjam novos requisitos, pode-se retornar aos passos anteriores para correções ou adições de novos requisitos e, novamente, executar a aplicação.

A Figura 71 mostra a execução da aplicação que Registra o Implante de Marcapasso em Pacientes.

Figura 71- Passo *Executar Aplicação*

A Figura 72 mostra a tela principal que registra o implante de marcapasso em um paciente.

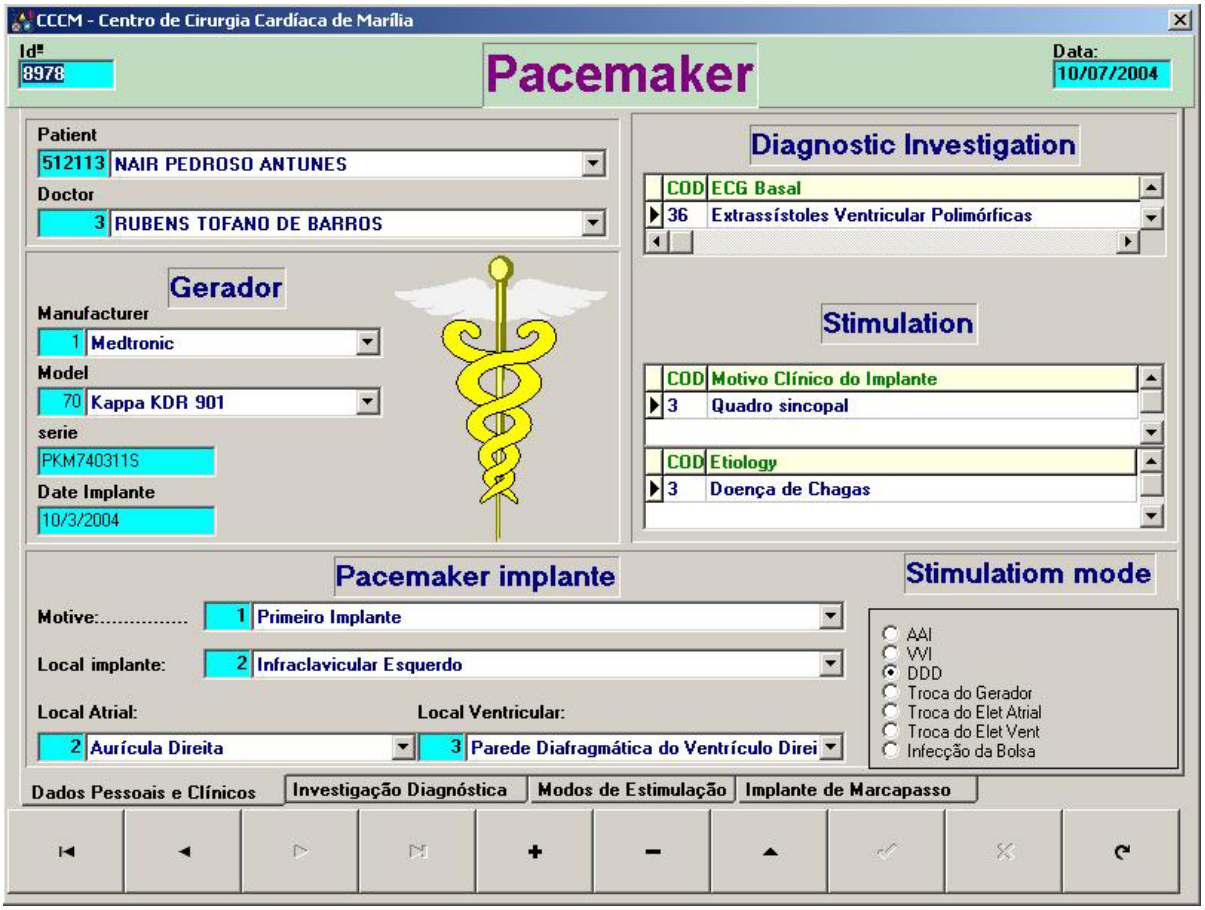


Figura 72– Tela Principal do Passo *Executar Aplicação*

Após a execução da aplicação o engenheiro de software pode refinar a aplicação, no ambiente *Delphi*, levando em consideração os requisitos não funcionais da aplicação.

5.4 Refinar Aplicação

Conforme mostra Figura 73, no passo *Refinar Aplicação***,** o engenheiro de software faz alterações no código gerado refinando-o até obter o código final da aplicação, que será executado no ambiente *Delphi*. Neste passo, o engenheiro de software importa o código gerado, no ambiente *Delphi*, para executá-lo. No *Delphi*, o código é reunido em um projeto da aplicação para facilitar seu gerenciamento.

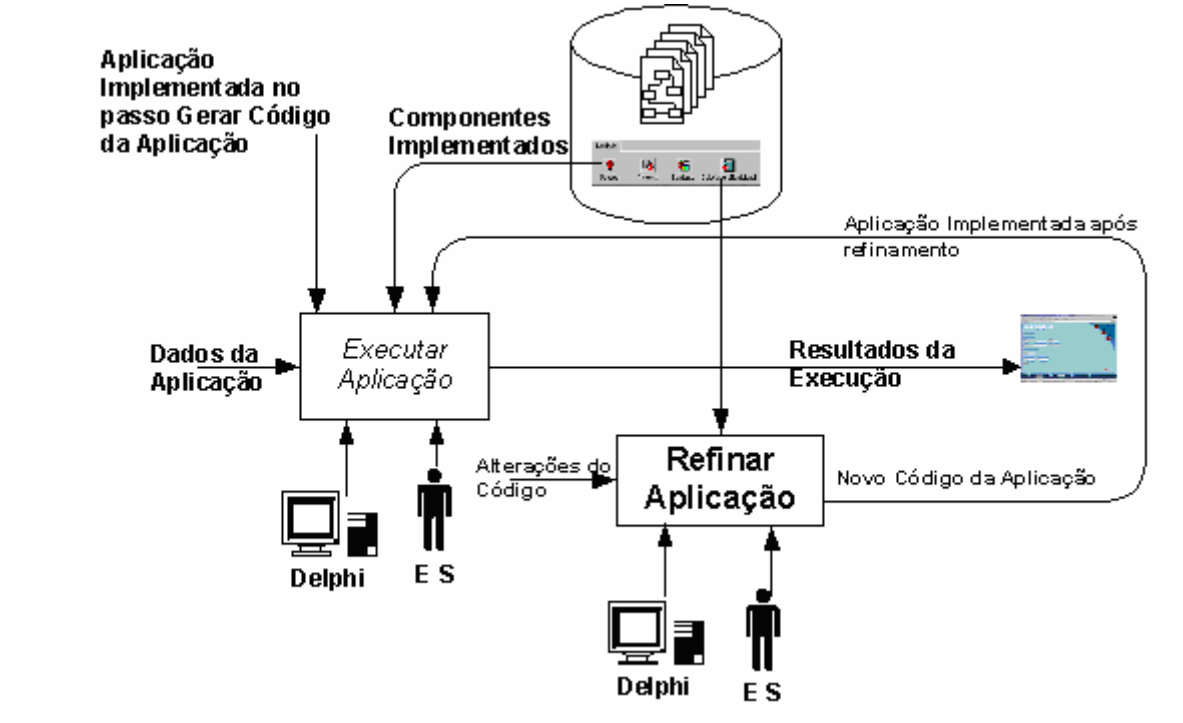


Figura 73– Passo *Refinar Aplicação*

5.5 Considerações Finais

Este capítulo apresentou a etapa de reutilização de componentes com implementação em *Delphi*, mostrando todo o seu processo e os principais artefatos gerados em cada passo.

Inicialmente o engenheiro de software modela a aplicação, a partir dos requisitos da aplicação, considerando os modelos dos componentes disponíveis para reuso, obtendo o projeto da aplicação.

Em seguida o engenheiro de software passa a gerar o código da aplicação, utilizando o ST Draco-PUC como mecanismo para realizar as transformações, e executa a aplicação gerada no ambiente *Delphi*. Neste passo entra-se com dados que permitem testar a aplicação construída. O engenheiro de software refina o código da aplicação para atender requisitos que incluem o tratamento de exceções, interfaces GUI (*Graphic User Interface*) e acesso a banco de dados.

A aplicação desenvolvida serviu para testar a etapa de reutilização dos componentes do domínio de cardiologia e validar os componentes construídos. Outras duas aplicações foram desenvolvidas e encontra-se em fase de testes no Instituto do Coração de Marília (ICM) e no Centro de Cirurgia Intervencionista de Marília (CCIM).

Capítulo 6

Avaliação

6.1 Considerações Iniciais

A automatização do processo de desenvolvimento de software pode ser facilitada com o uso da reutilização de componentes. Assim sendo, não somente a aplicação, mas todos os produtos intermediários gerados (documentação, *scripts* de criação de tabelas, código fonte) são considerados artefatos, independentemente de serem entregues ao usuário final.

A etapa de construção de componentes da abordagem proposta foi testada para o domínio de cardiologia, tendo em vista as experiências deste pesquisador no desenvolvimento de software para a área de saúde, mais especificamente para área de cardiologia. Foram construídos 357 componentes do domínio de cardiologia, conforme mostra a tabela 8 [ICM, 2004].

Tabela 8: Componentes construídos por área de assunto

Área de Assunto	Quantidade
Surgery	72
Pacemaker	40
Testes	96
Hemodynamic	52
Ecocardiograma	85
Appointment	12
TOTAL	357

A etapa de reutilização foi testada em 3 (três) sistemas do domínio de cardiologia. O sistema desenvolvido para o estudo de caso foi testado para no Centro de Cirurgia Cardíaca de

Marília (CCCM), sediado em Marília-SP. Este sistema foi modelado reutilizando 26 (vinte e seis) dos componentes construídos da área de assunto sobre "*Pacemaker*", visando registrar um implante de marcapasso em um paciente [CCCM, 2004].

Outro sistema desenvolvido foi para o Centro de Cirurgia Intervencionista de Marília (CCIM), sediado em Marília-SP, reutilizando 42 (quarenta e dois) dos componentes construídos da área de assunto "*Hemodynamic*", visando registrar as angioplastias realizadas nos pacientes encaminhados para a clínica [CCIM, 2004].

Mais um sistema foi desenvolvido para testar a reutilização dos componentes para o Instituto do Coração de Marília (ICM), reutilizando 32 (trinta e dois) dos componentes construídos da área de assunto "*Ecocardiograma*" e 6 (seis) dos componentes construídos da área de assunto "*Appointment*", visando registrar o agendamento de consultas de pacientes para a realização de exames de Ecocardiograma. Estes sistemas encontram-se em fase de testes e avaliação.

Como resultados intermediários da aplicação da abordagem proposta, nos 03 (três) sistemas, foram obtidos resultados considerados satisfatórios. Os 03 (três) sistemas foram desenvolvidos baseado na arquitetura da VCL do ambiente *Delphi* e rodam na rede local com o banco de dados relacional *Firebird*.

6.2 Análise dos Resultados

Os resultados obtidos mostraram a viabilidade da abordagem proposta. Foi possível construir uma grande quantidade de componentes do domínio de cardiologia que podem ser reutilizados em aplicações deste domínio.

Os componentes construídos estão disponíveis em uma biblioteca para reuso também no desenvolvimento de novos sistemas. Acredita-se que, com o desenvolvimento de novos sistemas do mesmo domínio poderá contribuir para refinar e melhorar ainda mais os componentes construídos.

Observou-se também que o tempo gasto para desenvolver uma aplicação reutilizando estes componentes é significativamente menor do que se ela fosse desenvolvida sem os componentes, já que não há a necessidade de construir os mesmos componentes diversas vezes. O sistema do Instituto do Coração de Marília - ICM foi desenvolvido a partir do sistema já existente, o SisCardio, e observou-se que a quantidade de linhas de código

diminuiu em cerca de 40%, uma vez que os componentes encapsulam os códigos dos métodos relacionados às regras de negócio e de persistência dos componentes.

Outra importante observação é que, após a reutilização ter sido realizada em diferentes sistemas, um mesmo componente pode vir a sofrer subseqüentes alterações, fazendo com que as aplicações previamente construídas não mais sejam compatíveis com as novas versões dos componentes. Sendo assim, é importante que se realize um controle das diferentes versões de um mesmo componente, de modo que o processo de manutenção e evolução das aplicações possa ser controlado de forma eficiente.

Capítulo 7

Conclusão

7.1 Considerações Iniciais

A utilização de componentes no desenvolvimento de software vem ganhando destaque, principalmente na indústria de software, onde existe uma grande preocupação em reduzir custos e aumentar a produtividade.

Este projeto desenvolve uma abordagem para a construção e reutilização de componentes de software com implementação em *Delphi*, que contribui na redução de custos e melhoria de produtividade. Além disso, os componentes podem ser reutilizados, diminuindo a redundância de código e facilitando as construções das aplicações de um domínio, cujas manutenções ficam mais simples e mais confiáveis, uma vez que os componentes foram previamente construídos e testados.

A arquitetura dos componentes segue um padrão da VCL do ambiente *Delphi*, que é uma hierarquia de classes, escritas em *ObjectPascal*, que suporta o desenvolvimento de aplicações através da reutilização. Um dos motivos em adotar o *Delphi* na abordagem foi devido à sua grande divulgação e utilização na indústria de software.

7.2 Principais Contribuições

Embora existam outros métodos para desenvolvimento de software, as vantagens oferecidas pela abordagem vêm do uso da automação de grande parte do processo de geração

de código, usando a ferramenta *MVCASE* e o ST Draco-PUC. Outras contribuições deste trabalho são:

- A integração de diferentes tecnologias, automatizando grande parte das tarefas do Engenheiro de Software, contribuindo para redução do tempo e custos do desenvolvimento de software, e produzindo um software mais organizado e fácil de ser mantido;
- A automatização de grande parte das tarefas do Engenheiro de Software possibilita a redução do tempo e custos de desenvolvimento de um sistema, através do reuso nos diferentes níveis de abstração, desde os requisitos, análise e projeto, até a implementação;
- Com base nas experiências no desenvolvimento de software para a área de Cardiologia, deste pesquisador, foi possível utilizar esta abordagem para a construção de componentes do domínio de cardiologia, contribuindo com a padronização de componentes da área de saúde do Consórcio de Componentes de Software (CCS-SIS);
- Outras aplicações, semelhantes à do estudo de caso, foram desenvolvidas para testar os componentes e a abordagem proposta. Com isto foi possível verificar a viabilidade da abordagem, que contribui para tornar mais confiável todo o processo de desenvolvimento de software baseado em componentes;
- Utilização do método de DBC Catalysis para a modelagem do componentes;
- Criação do domínio *ObjectPascal* no ST Draco-PUC, a partir da definição da gramática do *Delphi*;
- Criação do Transformador *MDLToObjectPascal* para geração do código dos componentes construídos e de suas aplicações;
- Extensão da ferramenta MVCASE para geração de código na linguagem *ObjectPascal*, a partir da definição de *stereotypes* para o ambiente *Delphi;*
- Extensão da ferramenta MVCASE para a geração do modelo de dados para obtenção dos scripts SQLs dos componentes persistentes; e
- A própria abordagem proposta, que define as etapas para a criação de componentes e a reutilização de componentes com implementação em *Delphi*;

Além das ferramentas RAD existentes no mercado, como *JBuilder*, C++*Builder*, e *Delphi*, várias outras são utilizadas na área de Desenvolvimento Baseado em Componentes.

Estas diferem da abordagem apresentada porque são mais direcionadas para a implementação e não para modelagem dos componentes.

Diferentes abordagens têm sido utilizadas na área de DBC, como o *Rational Unified Process* (RUP) e *UML Components*. A ferramenta *Rational Rose* tem sido usada como principal mecanismo para execução destes processos. Contudo, embora tanto o RUP como *UML Components*, tratem da implementação dos componentes, estas não tratam com detalhes esta etapa do ciclo de vida do componente, como no caso da abordagem proposta.

A *CASE Bold*, é outro exemplo, de ferramenta voltada para a construção de componentes dirigidas pelo modelo (*Model Driven Application - MDA*). Contudo, diferente da abordagem proposta, que permite utilizar diferentes linguagens de implementação devido à capacidade do ST Draco, e a ferramenta Bold atualmente tem versões apenas para *Delphi* e C++.

Existem também vários ambientes integrados para desenvolvimento de aplicações (IDE - *Integrated Development Environment*), como, por exemplo, o *Eclipse*, utilizado para modelagem de componentes, contudo o ambiente está centrado na linguagem Java.

Em resumo, a abordagem proposta diferencia dos trabalhos citados por:

- Combinar diferentes tecnologias, automatizando o processo de geração de código, através da ferramenta MVCASE e do ST Draco;
- Permitir trabalhar com diferentes domínios suportados pelo ST Draco;
- Possibilitar que os componentes construídos e implementados na linguagem *ObjectPascal* sejam reutilizados, tanto na ferramenta MVCASE, a partir dos seus modelos disponíveis, como no ambiente *Delphi*, a partir dos componentes disponíveis nas paletas; e
- Permitir uma integração e consistência entre os modelos dos componentes, com suas implementações na linguagem alvo da implementação, como no caso do *ObjectPascal*.

A abordagem dá mais um passo na automatização de grande parte das tarefas do Engenheiro de Software, o que pode contribuir na redução do tempo e custos do desenvolvimento de software.

7.3 Trabalhos Futuros

Com base nos resultados obtidos, destacam-se algumas idéias que visam dar continuidade a este trabalho:

- Suporte para o controle de versões, permitindo que o gerenciamento das diferentes versões dos componentes construídos facilite o processo de manutenção e evolução das aplicações. O grupo de Engenharia de Software do DC – UFSCar já possui trabalhos em andamento visando esse suporte;
- A utilização de outras linguagens alvo, além do *ObjectPascal*, para testar a abordagem proposta;
- A criação de um transformador *ObjectPascalToMDL* para permitir a engenharia reversa para a abordagem proposta, bem como a realização de reengenharia para sistemas desenvolvidos em *ObjectPascal* para obter seus modelos em MDL na ferramenta MVCASE;
- Melhorar as transformações responsáveis pela geração de código das aplicações, possibilitando a mínima intervenção do Engenheiro de Software no refinamento das aplicações;
- Aprimorar as extensões inseridas na ferramenta MVCASE com novas funcionalidades, de modo a permitir que possam ser modeladas aplicações para Web reutilizando os novos componentes disponíveis na versão 8 do ambiente *Delphi*;
- Por fim, sugere-se a realização de novos estudos de caso visando aumentar a confiabilidade dos componentes construídos.

Bibliografia

[Alexander, 1977] ALEXANDER, C. **A Pattern Language**. Oxford University Press, 1977.

[Arango, 1994] Arango, G.; **Domain Analysis Methods;** em Software Reusability; W. Schäfer, R. Prieto-Díaz, M. Matsumoto (ed.); cap. 2, Ellis Horwood, 1994.

[Booch, 1994] BOOCH, G. **Object-Oriented Analysis and Design with Applications**. Benjamin/Cummings, 1994.

[Booch, 1999] BOOCH, G. et al. **The Unified Modeling Language** – User Guide. USA: Addison Wesley, 1999.

[Booch, 2000] BOOCH, G. **UML Guia do Usuário**. Editora Campus, 2000.

[Bosch, 1997] BOSCH, J. **Adapting Object-Oriented Components**. In: ECOOP, 1997, Jyväskylä. Proceedings [S.l.]: Springer-Verlag, 1997.

[Borland, 2001] BORLAND/INPRISE. **Programming with Delphi** 2001.

[Borland, 2002] BORLAND/INPRISE. **Visual Component Library Reference** - 2002.

[Borland, 2002a] BORLAND/INPRISE. **Object Pascal Reference**. - 2002.

[Braga, 2000] BRAGA, R. **Busca e Recuperação de Componentes em Ambientes de Reutilização de Software.** Dissertação de Doutorado. Universidade Federal do Rio de Janeiro dezembro de 2000.

[Buschmann, 1995] BUSCHMANN, F. et al. **A System of Patterns**. Chichester: John Wiley & Sons, 1995.

[Castor, 2001] JaTS**: A Java Transformation System (2001)**. Castor, Fernando. Disponível: site URL: http://citeseer.ist.psu.edu/698789.html . Consultado em 10/10/2001.

[Catalysis, 2001] CATALYSIS. **Catalysis Enterprise Components.with UML**. Disponível: site URL: http://www.catalysis.org . Consultado em 10/08/2001.

[CBSE, 2002] **5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly**, In conjunction with the 24th International Conference on Software Engineering (ICSE), May 2002.

[CCCM, 2004] CCCM. **Centro de Cirurgia Cardíaca de Marília.** Disponível: site URL http://www.icm.com.br/infger/index.htm . Consultado em 22/07/2004 .

[CCIM, 2004] CCIM. **Centro de Cirurgia Intervencionista de Marília.** Disponível: site URL http://www.icm.com.br/infger/index.htm .Consultado em 22/07/2004.

[CDB-HQ, 2002] CDB-HQ. **Component-Based Development – Headquarters**. Disponível: site URL: http://www.cbd-hq.com/ . Consultado em 20/10/2002.

[Cheesman, 2000] Cheesman, J., Daniels, J. **UML Components: A Simple Process for Specifying Component-Based Software**. Addison-Wesley. USA, 1nd edition, 2000.

[CIS-UNIFESP, 2002] CIS-UNIFESP. **Centro de Informática em Saúde da Universidade Federal de São Paulo.** Disponível: site URL http://www.unifesp.br .Consultado em 22/02/2002.

[Coleman, 1994] COLEMAN, D. et al. **Object-Oriented Development – The Fusion Method**. Prentice Hall, 1994.

[COMET, 1997] Solberg, A., Berre, A., J. **Component Based Methodology Handbook**. Norway: SINTEF, 1997.

[DATASUS, 2000] DATASUS**. Departamento de Informática do Sistema Único de Saúde.** Disponível no site URL: http://www.datasus.gov.br/ . Consultado em 22/02/2002.

[D'Souza, 1999] D'SOUZA, D.; WILLS, A. Objects, **Components and Frameworks with UML – The Catalysis Approach.** USA:Addison Wesley, 1999.

[Donald, 1993] Donald, G. Firesmith. **Frameworks: the golden path to object Nirvana.** Journal of Object-Oriented Programming, 6(6):5-8, October 1993.

[Fayad, 1997] FAYAD, M.; SCHIMIDT, D. **Object-Oriented Application Frameworks.** Communications of the ACM, New York, v.40, n.10, p71-77, Oct. 1997.

[Fontes, 2001] FONTES, D.S, Bianchini, C. P., Prado, A. F., Sant'Anna, **Development of Applications for Information Research using Software Agents**, M., LAPTEC' 2001 Segundo Congresso de Lógica Aplicada à Tecnologia, Brasil, 12-14 de Novembro de 2001, Volume II, pág. 51-58.

[Fowler, 1999] FOWLER, M. **UML Destilled. Applying the Standard Object Modeling Language.** England:Addison Wesley, 1999.

[Gamma, 1995] GAMMA, E. et al. **Design Patterns. Elements of Reusable Object-Oriented Software.** USA: Addison-Wesley,1995.

[Griss, 1998] Griss, M., L., et. al, **Software Reuse: Nemesis or Nirvana?,** In Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Vancouver, British Columbia, Canada, 1998.

[HCSP-USP, 2002] HCSP-USP. **Hospital de Clínicas da Universidade de São Paulo**. Disponível: site URL http://www.hcnet.usp.br/ .Consultado em 22/02/2002.

[HCPA-UFRGS, 2002] HCPA-UFRGS. **Hospital de Clínicas de Porto Alegre**. Disponível: site URL http://www.hcpa.ufrgs.br/ .Consultado em 22/02/2002.

[Heineman, 1999] HEINEMAN. **An Evaluation of Component Adaptation Techniques**. International Workshop on Component Based Software Engineering. Los Angeles, maio 1999.

[ICM, 2004] ICM. **Instituto do Coração de Marília.** Disponível: site URL http://www.icm.com.br .Consultado em 22/07/2004.

[InCor, 2004] InCor. **Instituto do Coração do Hospital de Clínicas da Faculdade de Medicina de São Paulo.** Disponível: site URL http://www.incor.usp.br/ .Consultado em 22/02/2004.

[Jacobson, 2001] Jacobson, I., et. Al. **The Unified Software Development Process**. Addison-Wesley. USA, 4nd edition, 2001.

[Johnson, 1992] JOHNSON, R.; 1992. **Documenting Frameworks Using Patterns.** *Sigplan Notices – Trabalho Apresentado na OOPSLA*., New York, v.27, n. 10, Oct.

[Johnson, 1996] JOHNSON, R. **How to Develop Frameworks.** Tutorial Notes of ECOOP 96. Linz, Austria: July 1996.

[KobrA, 2000] Atkinson, C., et. al. **Component-Based Software Engineering:The KobrA Approach**, In 3rd International Workshop on Component-based Software Engineering: Reflection on Practice, in conjunction with the 22th International Conference on Software Engineering (ICSE). Limerick, Ireland, 2000.

[Kobryn, 2000]. KOBRYN, C. **Modeling Components and Frameworks with UML**. Communications of the ACM, Vol. 43 No 10, October 2000.

[Kruger, 1992] Krueger, C.; **Software Reuse**, ACM Computing Surveys, vol. 24, no 2, Junho 1992, pp. 131-183.

[Larman, 1999] LARMAN, C. **Utilizando UML e Padrões**. Prentice Hall , Inc, 1999.

[Leite, 1994] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA M. Draco-PUC Machine: **A Technology Assembly for Domain Oriented Software Development.** *3rd International Conference of Software Reuse*. IEEE Computer Society Press. In proceedings, pp. 94-100. Rio de Janeiro. 1994.

[Leite, 1995] Leite, Júlio Cesar S. P.; Prado, Antonio F.; Sant'Anna, Marcelo; Freitas, Felipe G. **O Uso do Paradigma Transformacional no Porte de Programas**

Cobol IX Simpósio Brasileiro de Engenharia de Software, pg 397-415, Recife, PE, Out 95.

[Leite, 1996] Leite, Júlio Cesar S. P.; Sant'Anna, Marcelo and Prado, Antonio F. **Porting Cobol Programs Using Transformational Aprroach** Journal of Software Maintenance: Reseach and Practice, vol 9, 3-31 , Out 1996 John Wiley&Sons Ltd.

[Lewis, 1995] LEWIS, T. et al. **Object Oriented Aplication Frameworks.** USA: Manning, 1995.

[Lucrédio, 2001] LUCRÉDIO, D., PRADO, A, F. **Ferramenta MVCASE – Estágio Atual: Especificação, Projeto e Construção de Componentes**, *XV Simpósio Brasileiro de Engenharia de Software*, *Sessão de Ferramentas*. Outubro de 2001.

[Moraes, 2001] Moraes, J. L. C., Prado, A. F. **Geração Automática de Código Delphi a partir de Especificações em Catalysis.** XXVII Conferência Latino Americano de Informática - CLEI'2001. ISBN: 980-11-0527-5. Mèrida, México. 24-28 de Setembro, 2001.

[Moraes, 2001a] Moraes, J. L. C., Prado, A. F. I Workshop de Informática Médica - **Framework de Cardiologia baseado em Componentes.** XV Simpósio Brasileiro de Engenharia de Software - SBES'2001. Rio de Janeiro- RJ, Brasil. 03-05 de Outubro, 2001.

[Moraes, 2002] Moraes, J. L. C.,Prado, A. F. **Desenvolvimento Baseado em Componentes de um Framework do Domínio de Cardiologia.** II Workshop Chileno de Ingeniería de Software, 2002, Copiapó-Chile.

[Moraes, 2002a] Moraes, J. L. C.,Prado, A. F. **Framework de Componentes do Domínio de Cardiologia.** In: The Second Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC/2002) - Sessão Técnica(4), Artigo nº1, 2002, Salvador – Brazil.

[Moraes, 2002b] Moraes, J. L. C., Bossonaro, A. A., Prado, A. F. **Desenvolvimento de um Framework, baseado em componentes, do domínio de Cardiologia.** VIII Congresso Brasileiro de Informática em Saúde, 2002, Natal-Brasil.

[Moraes, 2002c] Moraes, J. L. C., Prado, A. F. **Reutilização de Componentes do Framework do Domínio de Cardiologia (FrameCardio).** In: II Workshop de Informática Médica (WIM'2002) - Sessão Técnica(4), Artigo nº3.

[Moraes, 2002d] Moraes, J. L. C., Prado, A. F. **Automatic ObjectPascal Code Generation from Catalysis Specifications.** CLEI Eletronic Journal, Dez 2002.

[Moraes, 2003] Moraes, J. L. C., Prado, A. F. **Component-based Framework Development Process, of the Cardiology Domain**. CSIT'03, Mai 2003.

[Moraes, 2004] Moraes, J.L.C., Bossonaro, A., Garcia, V., Fontanette, V., Lucrédio, D., Prado, A.F., **An Approach for Construction and Reuse of Software Components *Frameworks* implemented in *Delphi***. Third IEEE International Symposium and School on Advance Distributed Systems - ISSADS 2004.

[Neighbors, 1984] NEIGHBORS, J.M. **The Draco approach to Constructing Software from Reusable Components.** *IEEE Transactions on Software Engineering*. v.se-10, n.5, pp.564-574, September, 1984.

[Orfaly, 1995]Edwards, Orfaly. **The Essential Distributed Objects Survival Guide**, chapter. 12, Object Frameworks: An Overview, pages221-238. John Wiley & Sons, 1995.

[Perspective, 2002] Select Perspective: Princeton Softech's practical methodology for delivering next generation applications. Disponível site The Active Archive Solutions Company, URL: http://www.princetonsoftech.com . Consultado em 10/10/2002.

[Pressman, 2001] PRESSMAN, R. S. **Software Engineering**. McGraw-Hill, 2001.

[Prieto-Diaz, 1990] Prieto-Diaz, R., Arango, G. **Domain Analysis and Software System Modeling**. IEEE Computer Society, Press Tutorial, 1990.

[PROCEMPA, 2002] PROCEMPA. **Processamento de Dados de Porto Alegre**. URL: http://poa.procempa.com.br/default.htm . Consultado em 22/02/2002.

[PRODABEL, 2002] PRODABEL. **Processamento de Dados de Belo Horizonte**. URL: http://www.ouro.pbh.gov.br/pbh/index.html . Consultado em 22/02/2002.

[Ralph, 1998] RALPH, E. JOHNSON and BRIAN FOOTE. **"*Designing reusable classes.* Journal of Object-Oriented Programming"**, 1(2):22—35, Jun 1988.

[RESCUEWARE, 2004] RELATIVITY. **Relativity Technologies**. URL: http://www.relativity.com/pages/home.asp . Consultado em 22/02/2004.

[Rogers, 1997] ROGERS, G. F. **Framework – Based Software Development in C++.** New Jersey:Prince Hall, 1997.

[Ross, 1977] ROSS, Douglas T. **Structured Analysis (SA): A language for communicating Ideas**, IEEE Transaction on Software Engineering, January 1977.

[Rumbaugh, 1991] Rumbaugh, R. **Object-Oriented Modeling and Design**. Prentice Hall, 1991.

[Rumbaugh, 1999] Rumbaugh, J., Jacobson, I., Booch, G. **The Unified Modeling Language Reference Manual**. Addison-Wesley, 1999.

[Sametinger, 1997] Sametinger, J., **Software Engineering with Reusable Components**, Springer, 1997.

[Sanches, 2000] SANCHES, Iolanda C., PRADO, Antonio F. **Framework para Desenvolvimento de Cursos para Web**. I Congresso de Lógica Aplicada à Tecnologia (LAPTEC). São Paulo, SP, Setembro de 2000.

[SBIS, 2004] SBIS. **Sociedade Brasileira de Saúde.** Disponível no site URL: http://www.sbis.org.br/ . Consultado em 10/06/2004.

[SEI, 1999a] Software Engineering Institute (SEI). **Domain Analysis**. Disponível site Software Engineering Institute (SEI), Engineering Practices, 1999, URL: http://www.sei.cmu.edu/domain-engineering/domain_anal.html . Consultado em 10/07/2002.

[SEI, 1999b] Software Engineering Institute (SEI). **Domain Design**. Disponível site Software Engineering Institute (SEI), Engineering Practices, 1999, URL:http://www.sei.cmu.edu/domain-engineering/domain_design.html . Consultado em 10/07/2002.

[Shapere, 1977] Shapere, D.; **Scientific Theories and Their Domains in the Structure of Scientific Theories** ; (Ed.) F.Suppe; Univeristy of Illinois Press, 1977.

[Simos, 1996] Simos, M. **Organization Domain Modeling (ODM): Domain Engineering as a Co-Methodology to Object-Oriented Techniques.** Fusion Newsletter, v.4 1996, Hewlett-Packard Laboratories, pp 13-16.

[Sun, 2002] **Java Technologies**. Disponível site Sun Microsystems, URL: http://java.sun.com/products. Consultado em 07/12/2002.

[Szyperski, 1998] Szyperski, Clements; **Component Software: Beyond Object-Oriented Programming**, Addison-Wesley, 1998.

[Taligent, 1999] TALIGENT. "**Leveraging object-oriented frameworks**". Taligent Inc. white paper. Disponível por http://www.taligent.com (18 mar. 2000).

[Tanaka, 1999] TANAKA, S. A. **Um Framework para Desenvolvimento de Gerenciadores de Workflow**. T.I. n. 857, Porto Alegre: CPGCC-UFRGS. Junho. 1999.

[Werner, 1998] WERNER, C.M.L., BRAGA, R., MATTOSO, M. **A Reuse Infrastructure Based on Domain Models**; *9th International Conference on Computing and Information; Winnipeg, Canada*, 1998, pp.227-234.

[Werner, 2000] WERNER, C.M.L et al. **Odyssey: Estágio Atual**, *XV Simpósio Brasileiro de Engenharia de Software SBES2000 – Sessão de Ferramentas* –Outubro, 2000.

[Werner, 2000a] WERNER, C.M.L, BRAGA, R. M.M. **Desenvolvimento Baseado em Componentes,** *XIV Simpósio Brasileiro de Engenharia de Software SBES2000 – Minicursos e Tutoriais* – pg. 297-329 – 2-6 de Outubro, 2000.

[Yoder, 1998] Yoder, J., W., Johnson, R., E., Wilson, Q., D. Connecting Business Objects to Relational Databases, In Pattern Languages of Programs (PLoP), Monticello, Illinois, USA,1998.