UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PACOTE DE COMPONENTES PARA CONTROLE DE OBJETOS EM DELPHI

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

RICARDO RAFAEL MEIER REICH

BLUMENAU, JUNHO/2003

2003/1 – 55

PACOTE DE COMPONENTES PARA CONTROLE DE OBJETOS EM DELPHI

RICARDO RAFAEL MEIER REICH

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Maurício Capobianco Lopes — Orientador na FURB

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

Prof. Maurício Capobianco Lopes

Prof. Everaldo Artur Grahl

Prof. Marcel Hugo

AGRADECIMENTOS

À minha mãe Ursula Meier pelo apoio, incentivo ao estudo, carinho e dedicação.

À minha noiva Patrícia Moro pelo amor, carinho e compreensão.

Ao professor Maurício que me orientou neste trabalho com paciência, dedicação e sabedoria.

Aos demais amigos e todas as pessoas que me apoiaram de forma direta ou indiretamente na elaboração deste trabalho.

A todos muito obrigado.

RESUMO

Este trabalho de conclusão de curso tem como objetivo apresentar um pacote de componentes para o ambiente Delphi para auxiliar na construção de sistemas baseados em objetos. Através deste pacote, o programador utiliza componentes de interface ligados aos atributos de classes implementadas em Object Pascal, conseguindo, com isso, aumentar sua produtividade no desenvolvimento de sistemas que utilizem essas classes.

ABSTRACT

This final research paper aims at developing a component package for the Delphi environment seeking the construction of systems based on objects, through the manipulation of the created classes in a project.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Uma das preocupações da indústria de software é a necessidade de criar software e sistemas corporativos muito mais rapidamente e a um custo mais baixo. De acordo com Martin (1995), para fazer bom uso da crescente potência dos computadores, precisa-se de um software de maior complexidade. Ainda que mais complexo, esse software também precisa ser mais confiável. A alta qualidade, portanto, é fundamental no desenvolvimento de software.

Em busca dessa alta qualidade as empresas desenvolvedoras de software utilizam cada vez mais as técnicas orientadas a objeto. Essas técnicas permitem que o software seja construído a partir de objetos que tenham um comportamento especificado. Os próprios objetos podem ser construídos a partir de outros, os quais, por sua vez, podem ser construídos ainda de outros.

Segundo Longo (1997), objetos são modelos que são criados no desenvolvimento dos softwares para representar entidades do mundo real. No dia a dia, vive-se cercado de objetos, como por exemplo carros, geladeiras, cadeiras, cães e gatos. Aplicações de software também possuem seus objetos que, assim como os do mundo real, têm tanto estados como comportamentos. Classes são moldes a partir dos quais criam-se objetos. Uma classe não é um objeto em si, mas somente a descrição de como serão os objetos de um determinado tipo quando instanciados a partir da especificação de uma classe. Objetos reais são obtidos instanciando-se classes pré-definidas. Pode-se instanciar vários objetos de uma determinada classe da mesma forma que se pode criar diversos bolos a partir de uma mesma receita.

No contexto do desenvolvimento de softwares com a tecnologia da orientação a objetos surgiram os chamados componentes de software. De acordo com Pressman (1995), componentes de software são “pedaços” de softwares gerados através de uma série de conversão que traduzem as exigências do cliente para uma linguagem de máquina executável no computador. Assim, programação baseada em componentes permite entre outras coisas, uma maior reusabilidade de código, além de fazer com que haja uma diminuição no tempo de construção dos softwares.

Como consta em Cantú (2002), a maioria das linguagens de programação modernas oferece suporte para a programação orientada a objetos, além de oferecerem diversos componentes de software. Segundo Cantú (2002), o ambiente de

desenvolvimento Delphi é um exemplo de ferramenta que alia as vantagens da programação orientada a objetos com os recursos oferecidos pelos componentes de software.

Entretanto, o que parece ser um contrasenso do ambiente, apesar do Delphi ser um ambiente baseado em objetos e componentes, ele não oferece componentes para manipular as classes que são criadas pelos desenvolvedores no decorrer do desenvolvimento do projeto, ou seja, os mesmos recursos oferecidos para o desenvolvimento de sistemas baseados em banco de dados não existem para sistemas baseados em classes e objetos.

Sendo assim, neste trabalho propõe-se construir um pacote de componentes a fim de facilitar, para o desenvolvedor da ferramenta Delphi, o controle de objetos, através das classes definidas no projeto.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um pacote de componentes que auxilie de forma dirigida e consistente a criação de uma interface para o controle de objetos (instâncias), tomando como base as definições das classes.

Os objetivos específicos são:

a) disponibilizar um pacote de componentes de controle de objetos para o ambiente Delphi;
b) facilitar o controle de objetos no ambiente Delphi.

1.2 ESTRUTURA DO TRABALHO

O segundo capítulo descreve sobre conceitos de orientação a objetos e componentes de software.

O terceiro capítulo apresenta a especificação, construção, implementação e funcionamento do protótipo.

No quarto capítulo apresentam-se as conclusões, limitações e extensões.

2 ORIENTAÇÃO A OBJETOS

Segundo Furlan (1998), orientação a objetos é como uma nova maneira de pensar os problemas utilizando modelos organizados a partir de conceitos do mundo real. O componente fundamental é o objeto que combina estrutura e comportamento em uma única entidade.

De acordo com Sommerville (2003), os sistemas orientados a objetos devem ser de fácil manutenção, uma vez que os objetos são independentes. A modificação da implementação de um objeto ou a adição de serviços não deve afetar outros objetos de sistemas. Devido ao fato de os objetos serem associados com coisas, existe sempre um nítido mapeamento entre as entidades do mundo real (como componentes de hardware) e seus objetos de controle no sistema. Isso melhora a facilidade de compreensão e, portanto, a facilidade de manutenção do projeto.

Os objetos são componentes potencialmente reutilizáveis porque são encapsulamentos independentes de estado e de operações. Os projetos podem ser desenvolvidos com o uso de objetos que tenham sido criados em projetos precedentes. Essa alternativa reduz os custos de projeto, programação e validação e também pode levar ao uso de objetos-padrão (o que, consequentemente, aumenta a facilidade de compreensão do projeto). Ela também reduz os riscos envolvidos no desenvolvimento de software.

Segundo Furlan (1998), a orientação a objetos apresenta também uma correspondência com o mundo real, visualizando objetos da natureza conforme são, individualizados e caraterizados com finalidade própria. Tais objetos permitem ser combinados ou utilizados separadamente, pois, em tese, apresentam baixa dependência externa e alta coesão interna de seus componentes, o que permite promover substituições sem afetar ou interferir com a operação dos demais objetos. Possibilita ainda incrementações graduais de componentes aos já instalados, ampliando a abrangência do sistema.

2.1 OBJETO

Segundo Sommerville (2003), um objeto é uma entidade que possui um estado e um conjunto definido de operações que operam nesse estado. O estado é representado por um conjunto de atributos de objeto . As operações associadas com o objeto fornecem serviços para outros objetos (clientes), que requisitam esses serviços quando alguma computação é necessária.

Os objetos são criados de acordo com uma definição de classe de objetos, que serve como um *template* para criar objetos. A classe apresenta declarações de todos os atributos e operações que devem ser associados a um objeto dessa classe.

De acordo com Jones (2001), os objetos servem a dois objetivos: facilitam a compreensão do mundo real e oferecem uma base real para a implementação em computador.

Todos os objetos tem identidade e são distinguíveis. O termo identidade significa que os objetos se distinguem por sua própria existência e não pelas propriedades descritivas que possam ter.

Segundo Boratti (2002), em termos de programação pode-se definir um objeto como sendo a abstração de uma entidade do mundo real, que apresenta sua própria existência, identificação, características de composição e que tem alguma utilidade, isto é, pode executar determinados serviços quando solicitado.

2.2 CLASSES

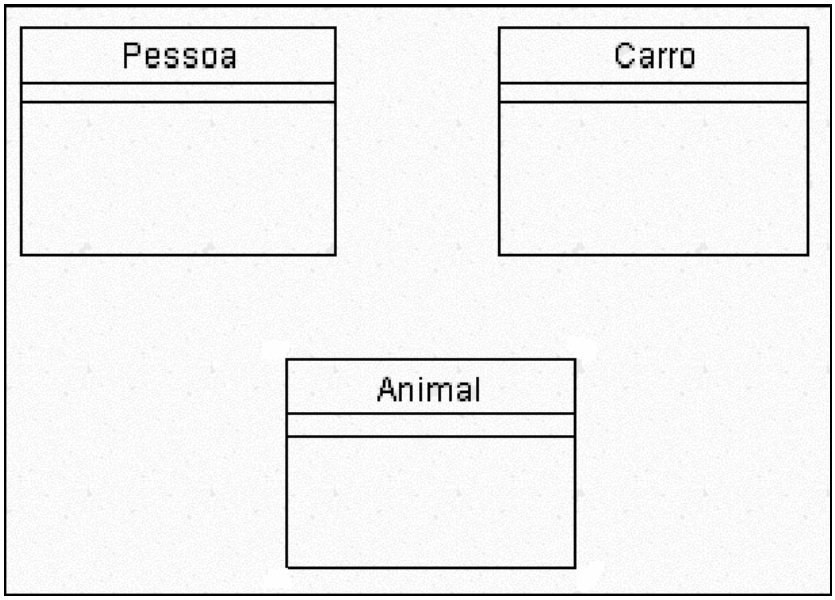
De acordo com Lemay (2001), uma classe é um modelo usado para criar um objeto. Todo objeto criado a partir da mesma classe terá recursos semelhantes, senão idênticos. As classes incorporam todos os recursos de um conjunto específico de objetos.

Ao escrever um programa em uma linguagem orientada a objetos, o desenvolvedor não define objetos individuais, em vez disso, define as classes usadas para criar esses objetos.

Segundo Rumbaugh (1998), uma classe de objetos descreve um grupo de objetos com propriedades semelhantes (atributos), o mesmo comportamento (operações), os mesmos relacionamentos com outros objetos e a mesma semântica.

A figura 2-1 mostra os símbolos equivalentes representativos de uma classe. Quando um objeto é gerado a partir de uma classe, ele adquire uma estrutura que é praticamente idêntica à essa classe.

FIGURA 2-1: SÍMBOLO DE UMA CLASSE



De acordo com Boratti (2002), uma classe define as características de um grupo de objetos, isto é, define como serão as instâncias pertencentes a ela. Assim, uma classe especifica quais serão os atributos dos objetos que pertencerem à mesma, bem como, os serviços que os objetos da classe poderão executar.

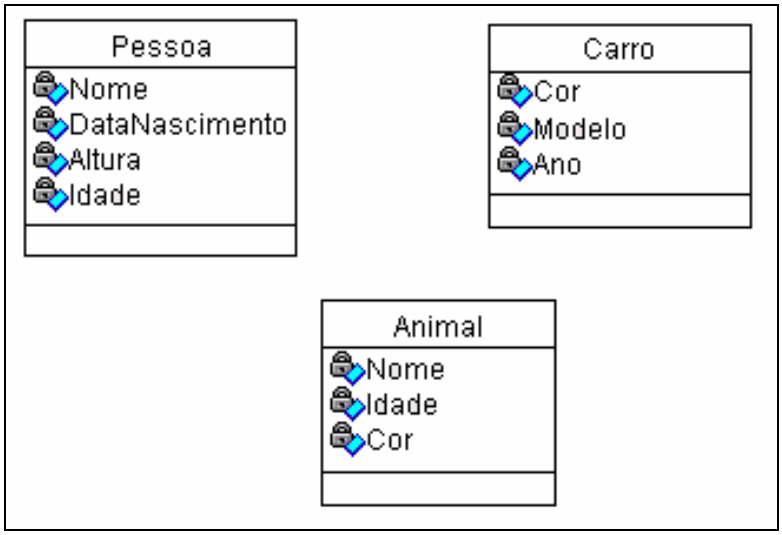
2.3 ATRIBUTOS

De acordo com Rumbaugh (1998), um atributo é um valor de dado guardado pelos objetos de uma classe. Nome, idade e peso são atributos dos objetos Pessoa. Cor, peso e ano-modelo são atributos dos objetos Carro. Cada atributo possui um valor para cada instância de objeto.

Os identificadores explícitos de objetos não são obrigatórios em um modelo de objetos. Cada objeto tem sua própria e única identidade. A maioria das linguagens baseadas em objetos gera automaticamente identificadores para referenciar objetos.

A classe Pessoa, na figura 2-2, tem três atributos listados: nome, datanascimento, altura e idade. Para a classe carro: cor, modelo e ano. Para a classe Animal: nome, idade e cor.

FIGURA 2-2: ATRIBUTOS

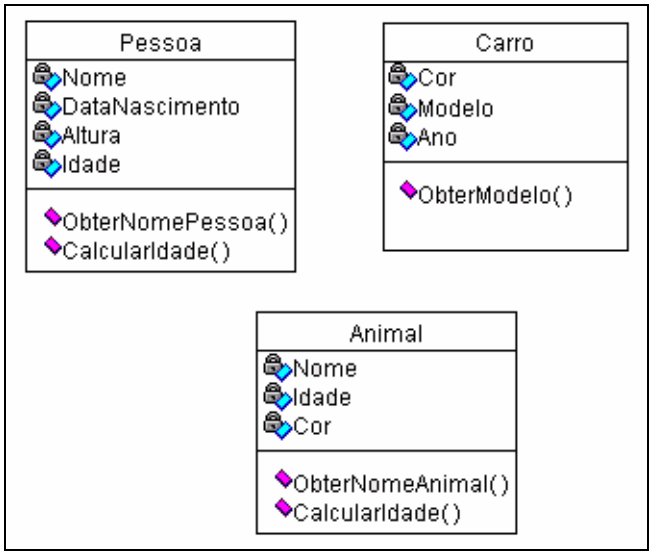


2.4 OPERAÇÕES E MÉTODOS

Segundo Martin (1995), métodos especificam a maneira pela qual os dados de um objeto são manipulados. Os métodos de um tipo de objeto referenciam somente as estruturas de dados desse tipo de objeto. Eles não devem acessar diretamente as estruturas de dados de outro objeto. Para usar a estrutura de dados de outro objeto, eles devem enviar uma mensagem a esse objeto.

Como mostrado na figura 2-3, as operações aparecem no compartimento mais baixo da estrutura. A classe pessoa possui dois métodos listados: ObterNomePessoa e CalcularIdade. Para a classe carro: ObterModelo. Para a classe animal: ObterNomeAnimal e CalcularIdade.

FIGURA 2-3: OPERAÇÕES



2.5 ENCAPSULAMENTO

Segundo Ambler (1998), quando determina-se o que uma classe sabe e faz, diz-se que abstrai-se a interface da classe. Quando se oculta como a classe vai alcançar isto, diz-se que é encapsulada. Quando se projeta bem as classes restringindo o acesso aos seus atributos, diz-se que suas informações são ocultadas.

Como consta em Martin (1995), encapsulamento é o ato de empacotar ao mesmo tempo dados e métodos. O objeto esconde seus dados de outros objetos e permite que os dados sejam acessados por intermédio de seus próprios métodos. Isso é chamado de ocultação de informações. O encapsulamento protege os dados de um objeto contra a adulteração.

Segundo Martin (1995), o encapsulamento é importante porque separa a maneira como um objeto se comporta da maneira como ele é implementado. Isso permite que as implementações do objeto sejam modificadas sem exigir que os aplicativos que as usam sejam também modificados.

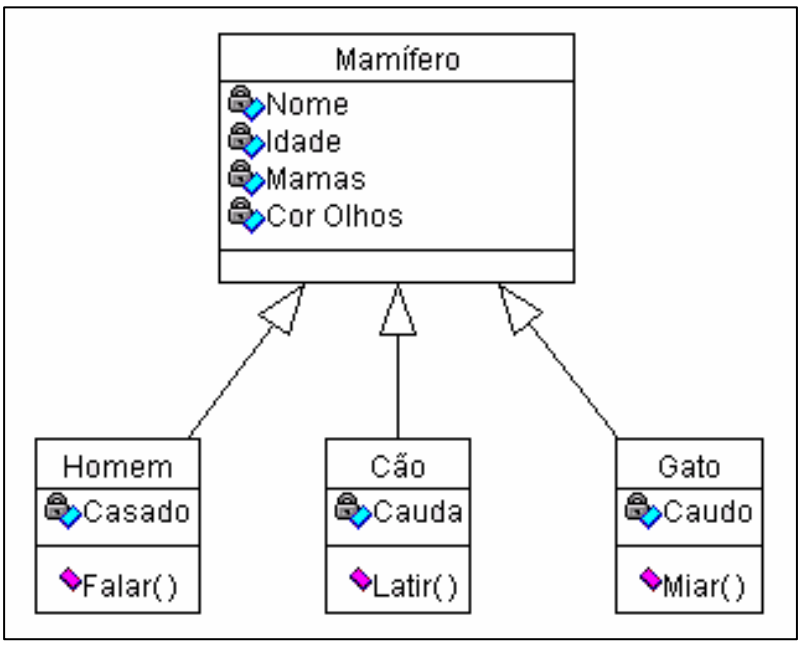
2.6 HERANÇA

Segundo Martin (1995), um tipo de objeto de alto nível pode ser especializado em tipos de objetos de níveis mais baixos. Um tipo de objeto pode ter subtipos. Há uma hierarquia de tipos de objeto, subtipos, subsubtipos e assim por diante.

De acordo com Ambler (1998), existem diversas similaridades entre diferentes classes. Com muita freqüência duas ou mais classes compartilharão os mesmos atributos e/ou os mesmos métodos. A herança é o mecanismo que evita escrever o mesmo código várias vezes. A herança modela relacionamentos do tipo “é” ou “é semelhante”, permitindo usar dados e códigos existentes.

A figura 2-4 mostra uma hierarquia de herança com uma herança simples (cada subclasse tem uma superclasse). Este exemplo mostra a criação da superclasse Mamífero e de três subclasses especializadas a partir dela: Homem, Cão e Gato.

FIGURA 2-4: HIERARQUIA DE HERANÇA SIMPLES



2.7 PERSISTÊNCIA

De acordo com Ambler (1998), a persistência descreve as questões de como salvar objetos para armazenamento permanente. Para tornar um objeto persistente, o desenvolvedor tem que salvar o valor de seus atributos em mídia de armazenamento permanente (disco rígido), assim como todas as informações necessárias para se manter os relacionamentos (tanto relacionamentos de agregação como instanciamento) com que ele esteja envolvido. A persistência permite que os objetos existam entre as execuções do sistema. Um objeto persistente tem que ter a capacidade de ser recriado de forma idêntica no futuro.

2.8 AGREGAÇÃO

Segundo Jones (2001), a agregação é uma construção familiar por meio da qual os sistemas de software representam estrutura da vida real. Em outras palavras, a agregação é uma associação grupo/membro.

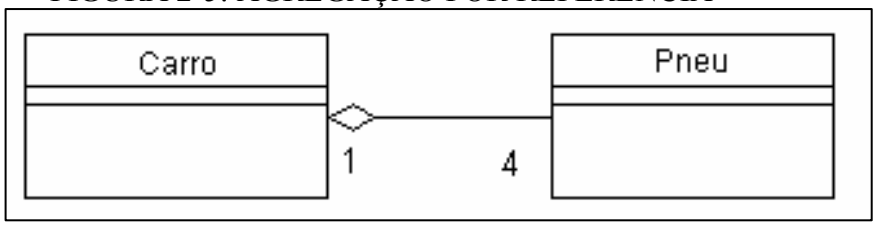
De acordo com Rumbaugth (1994), a agregação é um modo de associação forte na qual um objeto agregado é feito de componentes. Os componentes fazem parte do agregado. O agregado é, em termos semânticos, um objeto estendido tratado como uma unidade em muitas operações, embora fisicamente ele seja composto por objetos

menores. Um objeto agregado pode ter diversas partes. As partes podem ou não existir fora do agregado ou constar em muitos agregados. A agregação é inerentemente transitiva, ou seja, um agregado possui partes, que, por sua vez, podem ter outras partes.

Há dois tipos de agregação:

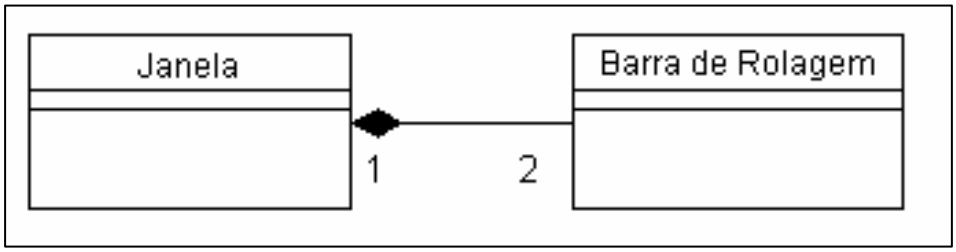
a) agregação por referência: Os objetos agregador e agregado possuem tempos de vida independentes. Por exemplo, quando o objeto carro (agregador) deixa de existir, os quatro objetos da classe pneu (agregado) podem continuar existindo (Figura 2-5).

FIGURA 2-5: AGREGAÇÃO POR REFERÊNCIA



b) agregação por valor (ou composição): Os objetos agregador e agregado estão fortemente acoplados e seus tempos de vida são dependentes. Quando a janela (agregador) deixa de existir, a barra de rolagem (agregado) também deixa de existir (Figura 2-6).

FIGURA 2-6: AGREGAÇÃO POR VALOR



2.9 POLIMORFISMO

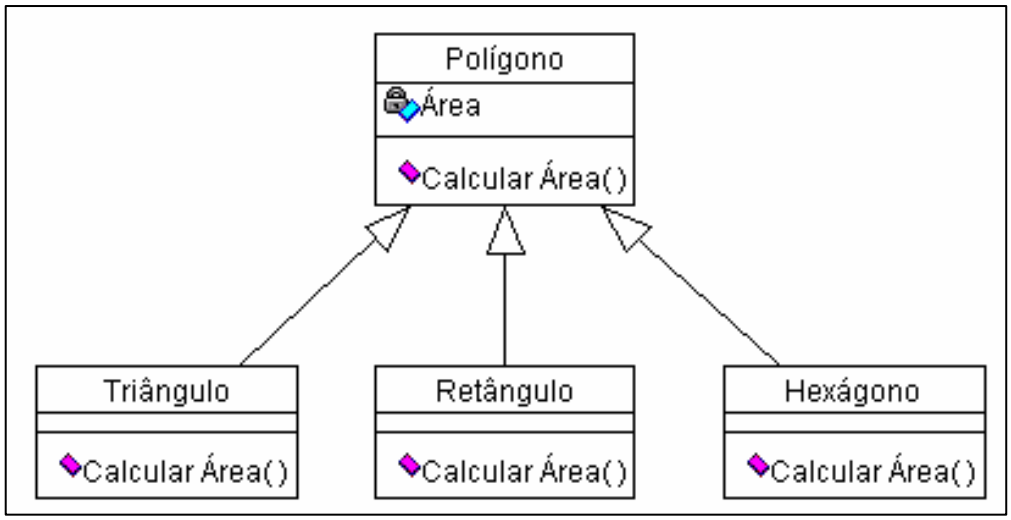
Segundo Jones (2001), a palavra polimorfismo origina-se de duas palavras gregas que significam, muitas e forma. Algo que é polimórfico, portanto, apresenta a propriedade de assumir muitas formas.

Como consta em Furlan (1998), tais formas se referem aos vários comportamentos que uma mesma operação pode assumir, assim como à capacidade de uma variável referir-se a objetos diferentes que preenchem certas responsabilidades dependendo da mensagem que lhes é passada.

De acordo com Ambler (1998), polimorfismo diz que um objeto pode tomar qualquer de várias formas e que outros objetos podem interagir com este objeto sem que saibam qual forma específica ele toma.

Na figura 2-7 as classes triângulo e retângulo hexágono tem operações denominadas calcularArea, assim como sua superclasse polígono. Essas operações realizam a mesma tarefa que a versão calcularArea da classe polígono: calculam a área da superfície total limitada pelo contorno, diferenciando-se apenas pela fórmula do cálculo tendo em vista que o cálculo da área das figuras é diferente.

FIGURA 2-7: POLÍGONO E SUAS TRÊS SUBCLASSES



2.10 COMPONENTES DE SOFTWARE

De acordo com McClure (2001), um componente de software pode ser definido como um módulo independente que provê informações sobre o que faz e como deve ser utilizado, ocultando a implementação. A interface identifica o componente, seus comportamentos e mecanismos de interação.

Entretanto não existe uma definição consensual para “componente de software”, o que é um indicativo da falta de maturidade da tecnologia. Não existe acordo em

relação ao que o componente é, exceto que os componentes são para efetuar composições. De acordo com Guerra (2000), um componente é uma unidade de composição com uma interface contratualmente especificada e apenas com dependências explícitas de contexto. Um componente pode ser instalado isoladamente e ser sujeito a composição por terceiros. Desta forma, componentes são produtos que são distribuídos com o objetivo de serem compostos por terceiros. Assim, os componentes devem ser construídos tão independentes de determinado contexto quanto possível para permitir sua especificação e integração. Quanto mais um componente é independente de um determinado contexto mais facilmente ele pode ser reutilizado em outros contextos. No entanto, um componente totalmente independente de qualquer contexto não pode ser reutilizado, uma vez que necessita de um contexto para poder ser integrado. Essa contradição constitui um dos maiores desafios no projeto de componentes reutilizáveis.

Segundo Guerra (2000), os componentes de software são um conceito que permite resolver alguns problemas da crise do software. Comparado com outras engenharias onde a utilização de componentes tem sido bem sucedida, a tecnologia dos componentes de software ainda está em formação. Os aspectos técnicos e em particular a interoperabilidade constituem a maioria das pesquisas nesta área. Do ponto de vista técnico, os componentes são fundamentalmente semelhantes aos objetos, quer nos objetivos quer nas soluções encontradas. A natureza específica dos componentes surge apenas no contexto da produção de software a um nível não exclusivamente técnico. Embora as alterações ao desenvolvimento de software sejam desencadeadas por inovações técnicas, o mercado dos componentes de software não se limita a problemas técnicos.

O êxito na reutilização de componentes depende de melhores técnicas de composição. A forma mais freqüente de descrever a informação de composição é através da interface do componente e alguma informação adicional, normalmente sob uma forma informal. Esta informação não é suficientemente detalhada para permitir resolver os problemas de interoperabilidade. Descrições semânticas restringem-se a alguns domínios de aplicação. Torna-se necessário, portanto, introduzir níveis intermediários entre as descrições sintáticas de interfaces e descrições semânticas completas. Uma solução pragmática consiste em documentar o componente com a informação necessária à sua integração durante o seu processo de desenvolvimento.

2.10.1 CARACTERIZAÇÃO DE COMPONENTES

De acordo com Guerra (2000), cada vez mais a criação de sistemas baseia-se em componentes já existentes. Desta forma torna-se cada vez mais importante dispor de uma correta caracterização de cada componente. Sem essa caracterização a compreensão e adaptação dos componentes serão prejudicadas. Alem disso, a caracterização de componentes é essencial à gestão de bibliotecas de componentes, que é uma exigência da engenharia de software baseada na reutilização sistemática de componentes. A caracterização de componentes é um passo no sentido de facilitar o aparecimento de tais aproximações, podendo ser vista como uma extensão da caracterização de objetos, oferecendo uma base para o desenvolvimento, gestão e utilização de componentes.

Os elementos estruturais são partes essenciais dos componentes pois são eles que dão corpo ao componente e, seguindo a convenção dos objetos, são chamados de atributos. Da mesma forma, as operações captam o comportamento dinâmico do componente, representando o serviço/funcionalidade que este oferece aos outros componentes que com ele interagem. Tal como no caso dos objetos, os atributos e as operações constituem os aspectos fundamentais de um componente.

Segundo Guerra (2000), a composição de componentes está sujeita a um conjunto de restrições. As restrições estruturais, onde certas composições de atributos não são permitidas, e as restrições operacionais, onde nem todas as seqüências de invocação de componente são possíveis. Os padrões estruturais e operacionais permitem definir as composições aceitáveis para determinado componente. O diagrama de transição de estados representa, no caso dos objetos, as seqüências de invocação possíveis de cada objeto.

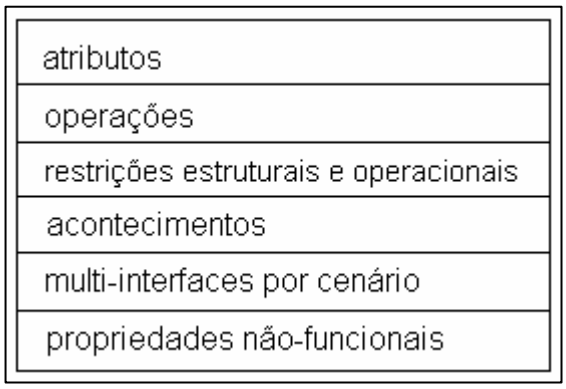
Um componente pode interagir com um certo número de outros componentes dependendo do papel, ou perspectiva, que ele representa. As interações entre os componentes podem variar de acordo com as perspectivas de cada um. Por exemplo, ao interagir com um certo tipo de componentes a partir de uma determinada perspectiva, apenas algumas operações e algumas restrições operacionais podem ser aplicadas. Este fato sugere a necessidade de definir para cada componente, protocolos de interação, ou interfaces, por cada perspectiva. Os cenários oferecem descrições de casos em que os componentes são vistos de diferentes perspectivas, o que facilita a especialização e

integração do componente. O ponto de vista do componente é mais atraente nas operações de busca e seleção, não havendo consenso quanto à solução a adotar.

De acordo com Guerra (2000), um outro aspecto de um componente diz respeito às suas propriedades não funcionais, tais como segurança, legibilidade, desempenho e confiabilidade. No contexto da construção de sistemas a partir de componentes existentes, o impacto no resultado final é particularmente importante. Algumas propriedades não funcionais não podem ser quantificadas, como a legibilidade.

A figura 2-8 apresenta as principais características de um componente:

FIGURA 2-8: CARACTERÍSTICAS



FONTE: Guerra (2000)

Contudo, é ainda difícil dispor de uma caracterização normalizada para os componentes, uma vez que mesmo os aspectos com origem na tecnologia dos objetos não estão totalmente estáveis no seu próprio domínio.

2.10.2 CATÁLOGO DE COMPONENTES

Segundo Guerra (2000), o desenvolvimento de aplicações baseadas em componentes requer a existência de catálogos de componentes de softwares. Para permitir construir softwares essencialmente baseados em componentes com origem em catálogos é necessário que, tanto os componentes quanto os catálogos, obedeçam a um conjunto de requisitos fundamentais:

a) os componentes do catálogo devem formar uma taxinomia sistemática: a taxinomia sistemática permite guiar o desenvolvedor da aplicação na busca e seleção dos componentes do catálogo. O desenvolvimento de um conjunto de

catálogos, cada um suportando uma funcionalidade bem definida, é essencial para a consolidação de um desenvolvimento de software baseado na reutilização de componentes. O sucesso de muitas linguagens de programação tem sido devido ao aparecimento de extensas bibliotecas de componentes normalizados para a linguagem;

b) os componentes devem ser genéricos: a principal motivação para a reutilização de software é a redução do esforço necessário para o seu desenvolvimento. Quanto mais vezes um componente for reutilizado, menor é o esforço total de desenvolvimento. Por outro lado, quanto mais genéricos forem os componentes, menos componentes são necessários para cobrir um dado domínio. Tal fato facilita a sua posterior seleção e aumenta a possibilidade de ser reutilizado;
c) os componentes devem ser eficientes: as bibliotecas devem procurar ter realizações razoavelmente eficientes. Para tal, basta ter presente as ordens dos algoritmos de ordenação ou de busca. Ignorar os requisitos de eficiência de uma forma arbitrária pode obrigar a redesenhar grande número de componentes;
d) os catálogos devem ser abrangentes: no primeiro requisito afirmava-se que os componentes do catálogo devem formar uma taxinomia sistemática num dado domínio. Para que o uso de bibliotecas de componentes seja difundido e vulgarizado é necessário que cada catálogo seja abrangente, ou seja, deve cobrir uma parte significativa da taxinomia do domínio. Se o catálogo for abrangente evita-se a produção doméstica, em que o programador opta por desenvolver software a partir do nada, apenas por desconhecer que existe algo diretamente disponível. O catálogo ao ser abrangente oferece garantias mínimas de conter pelo menos um componente que obedece minimamente aos requisitos;
e) os componentes devem residir em repositórios integrados: a existência de um repositório comum, com interfaces e formato de dados normalizados, permite eliminar dependências de hardware, sistema operacional, bases de dados ou interfaces gráficas. Do ponto de vista dos componentes, um repositório de componentes define um modelo para documentar, guardar e distribuir componentes de uma forma normalizada. Os componentes podem ter

informações associadas que permitem classificar, e posteriormente selecionar, tornando o processo de reutilização mais integrado e completo.

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo é apresentado o desenvolvimento do sistema, seus requisitos principais, bem como a especificação, implementação e operacionalidade.

3.1 REQUISITOS PRINCIPAIS DO SISTEMA

Este trabalho visa o desenvolvimento de um pacote de componentes para controle de objetos, tomando como base as classes definidas pelo programador de sistemas do ambiente Delphi. Para isto, deve-se oferecer ao programador uma paleta com uma diversidade de componentes de controle visual de objetos que possa interagir com as classes definidas no projeto.

A construção deste pacote de componentes exige a análise de uma *unit* que é definida pelo programador, buscando as classes e propriedades declaradas nela. Após a extração dessas classes e propriedades, devem ser criados componentes e métodos para o controle de instâncias de uma classe com as mesmas características da classe definida pelo programador.

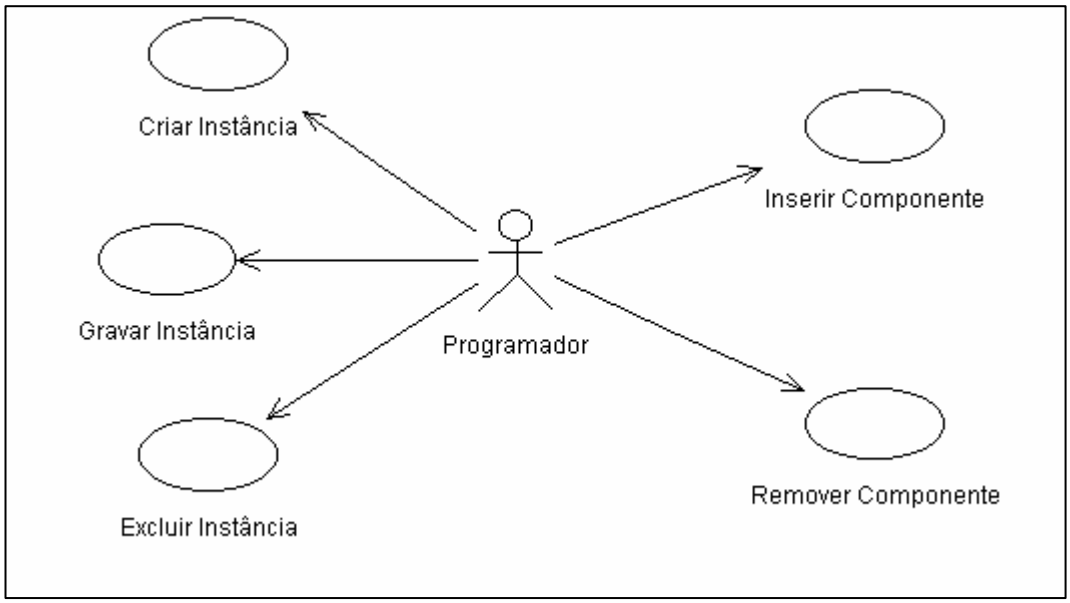
3.2 ESPECIFICAÇÃO

Para a especificação foi utilizada a ferramenta Rational Rose versão 4.0 Demo, utilizando a linguagem para modelagem de sistemas *Unified Modeling Languge* (UML). A seguir são apresentados os principais diagramas criados com auxilio da ferramenta Rational Rose.

3.2.1 DIAGRAMA DE CASOS DE USO

Na figura 3-1 é apresentado o diagrama de casos de uso.

FIGURA 3-1: DIAGRAMA DE CASOS DE USO



O ator Programador representa o programador do ambiente Delphi, que utilizará o pacote de componentes.

Os casos de uso são:

a) inserir componente: o programador insere os componentes do pacote no projeto, tendo como objetivo controlar os objetos das classes declaradas nele;

b) remover componente: o programador remove os componentes do pacote do projeto;

c) criar instância: o programador cria instâncias de uma das classes declaradas no projeto;

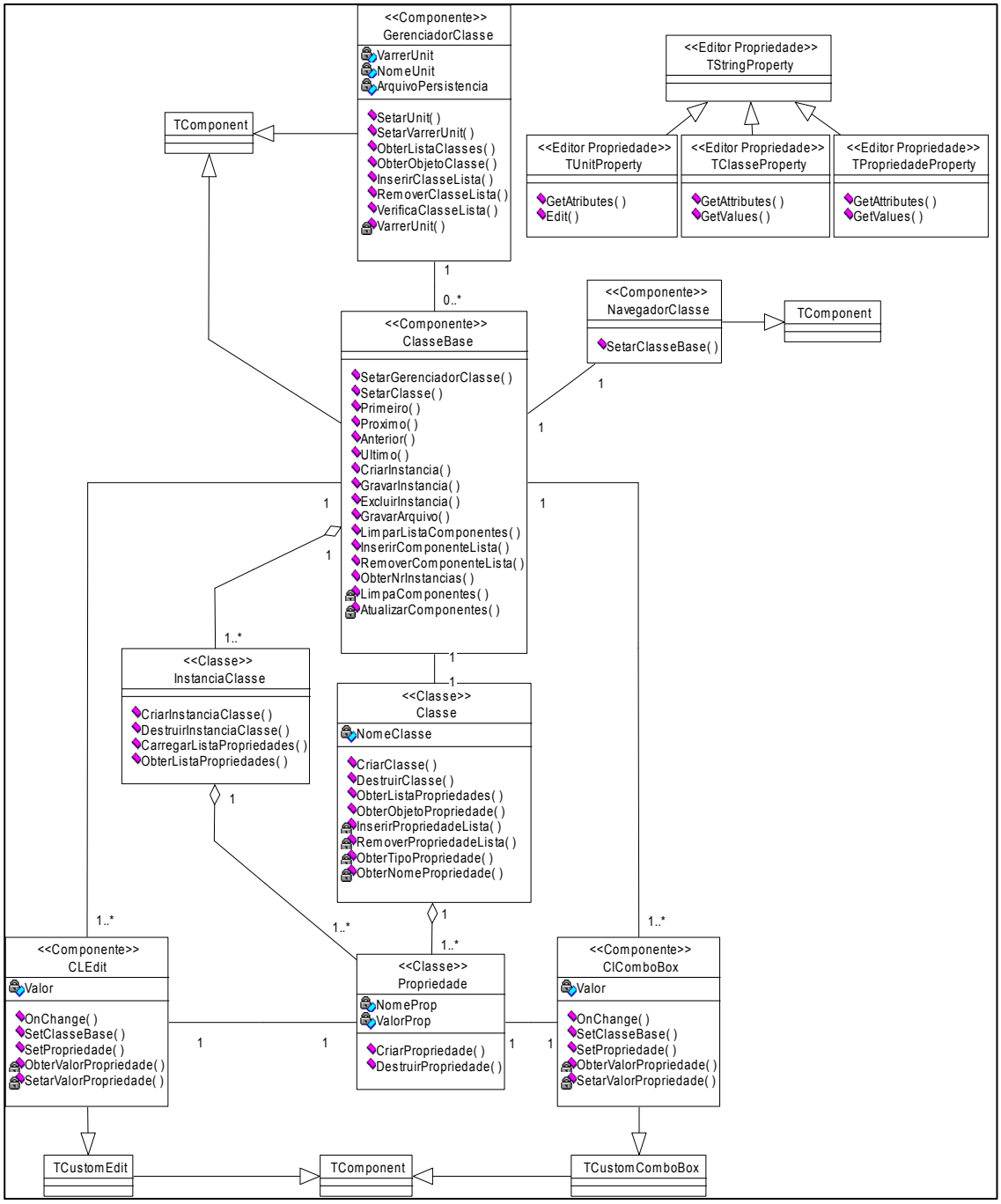
d) gravar instância: o programador grava as instâncias criadas de uma das classes declaradas no projeto;

e) excluir instância: o programador exclui as instâncias criadas de uma das classes declaradas no projeto;

3.2.2 DIAGRAMA DE CLASSES

Na figura 3-2 é representado o diagrama de classes para construção do protótipo.

FIGURA 3-2: DIAGRAMA DE CLASSES



No componente *GerenciadorClasse*, o programador define a *unit* que contém as classes e propriedades sobre as quais tem interesse em desenvolver sua aplicação. Este componente varre a *unit* carregando as classes encontradas na classe *Classe* e as propriedades na classe *Propriedade.* Ainda neste componente, o programador define o

nome do arquivo para gravar os dados das instâncias criadas a partir da classe selecionada.

No componente *ClasseBase*, o programador seleciona uma das classes carregadas pelo *GerenciadorClasse*. Este componente possui métodos para o controle das instâncias da classe *InstanciaClasse*. Esta classe possui as mesmas características da classe definida pelo programador no componente *ClasseBase*.

Os componentes *ClEdit* e *ClCombobox* são componentes visuais para o controle das instâncias da classe *InstanciaClasse*. Nestes componentes, o programador define a propriedade do da classe definida no componente *ClasseBase*, carregada pelo *GerenciadorClasse.*

A classe *Classe* armazena as classes que o componente *GerenciadorClasse* obteve a partir da *unit* definida pelo programador.

A classe *Propriedade* armazena as propriedades que o componente *GerenciadorClasse* obteve a partir da *unit* definida pelo programador.

O componente *NavegadorClasse* é um componente visual que faz a chamada dos métodos para o controle das instâncias da classe *InstanciaClasse*.

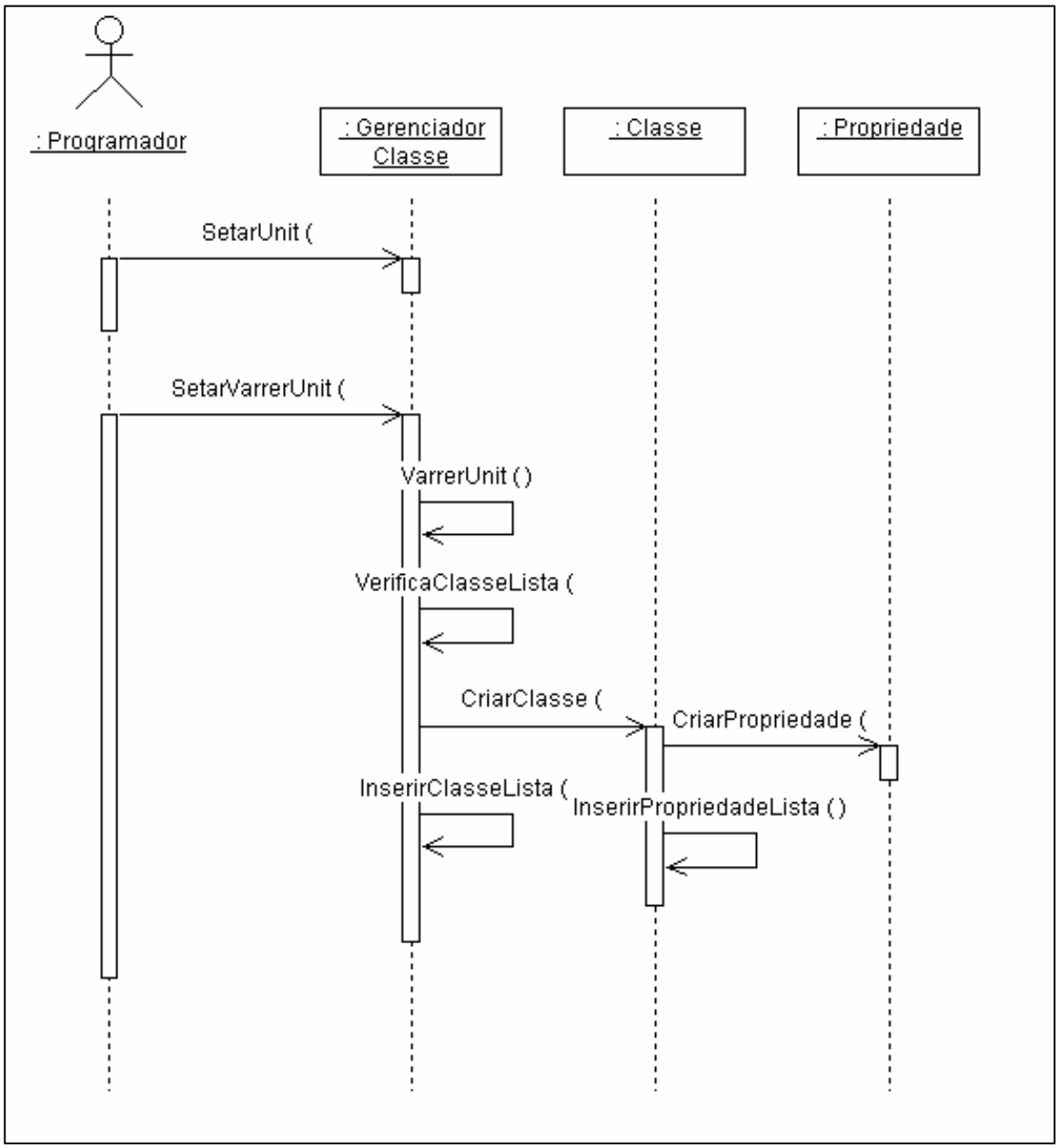
As classes *TUnitProperty, TClasseProperty e TPropriedadeProperty* são editores de propriedade do ambiente Delphi.

As classes *TComponent, TCustomEdit, TCustomCombobox* e *TStringProperty* são componentes próprios do ambiente Delphi.

3.2.3 DIAGRAMA DE SEQÜÊNCIA

A figura 3-3 apresenta o diagrama de seqüência para a inclusão do componente *GerenciadorClasse.*

FIGURA 3-3: DIAGRAMA DE SEQÜÊNCIA DE INCLUSÃO DO COMPONENTE GERENCIADORCLASSE

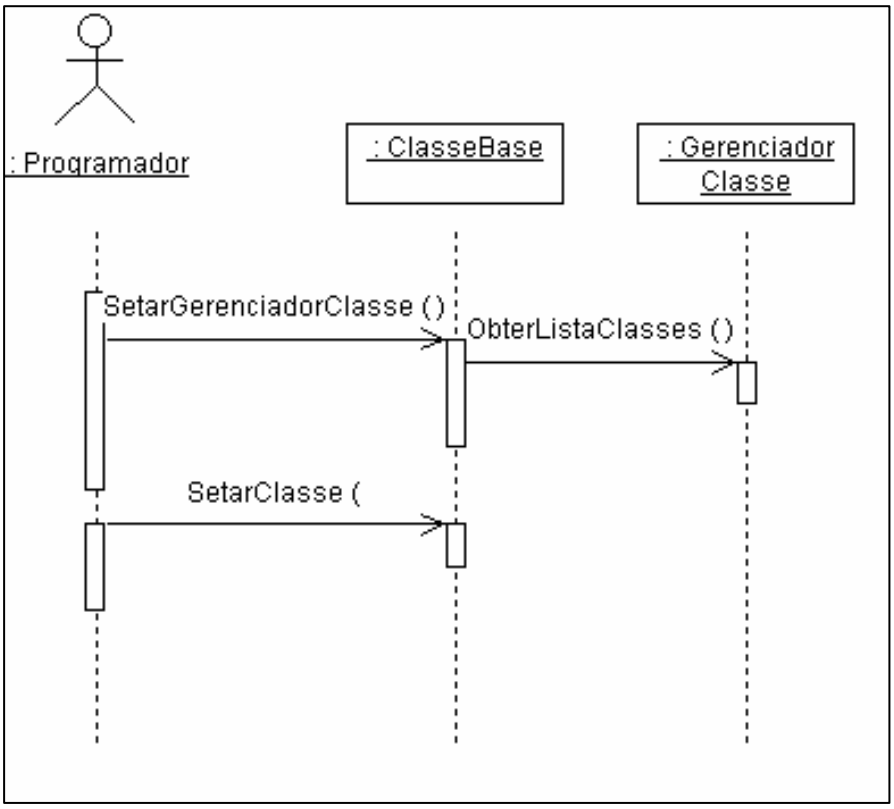


Na inclusão do componente *GerenciadorClasse*, o método *SetarUnit* permite ao programador a escolha da *unit* onde estão as classes do projeto. O método *SetarVarrerUnit* faz com que o método *VarrerUnit*seja chamado. O método *VarrerUnit* analisa a *unit* procurando as classes e propriedades contidas nela. Após encontrar uma classe, o método *VerificaClasseLista*é chamado, ignorando -a se a classe já tiver sido criada, caso ainda não, o método *CriarClasse* é chamado. Este método inclui a classe

encontrada na classe *Classe*. O método *CriarPropriedade* carrega as propriedades encontradas na classe *Propriedade*. O método *InserirPropriedadeLista* adiciona a propriedade encontrada na lista de propriedades da classe *Classe.* O método *InserirClasseLista* adiciona a classe encontrada na lista de classes do componente *GerenciadorClasse*.

A figura 3-4 apresenta o diagrama de seqüência para a inclusão do componente *ClasseBase.*

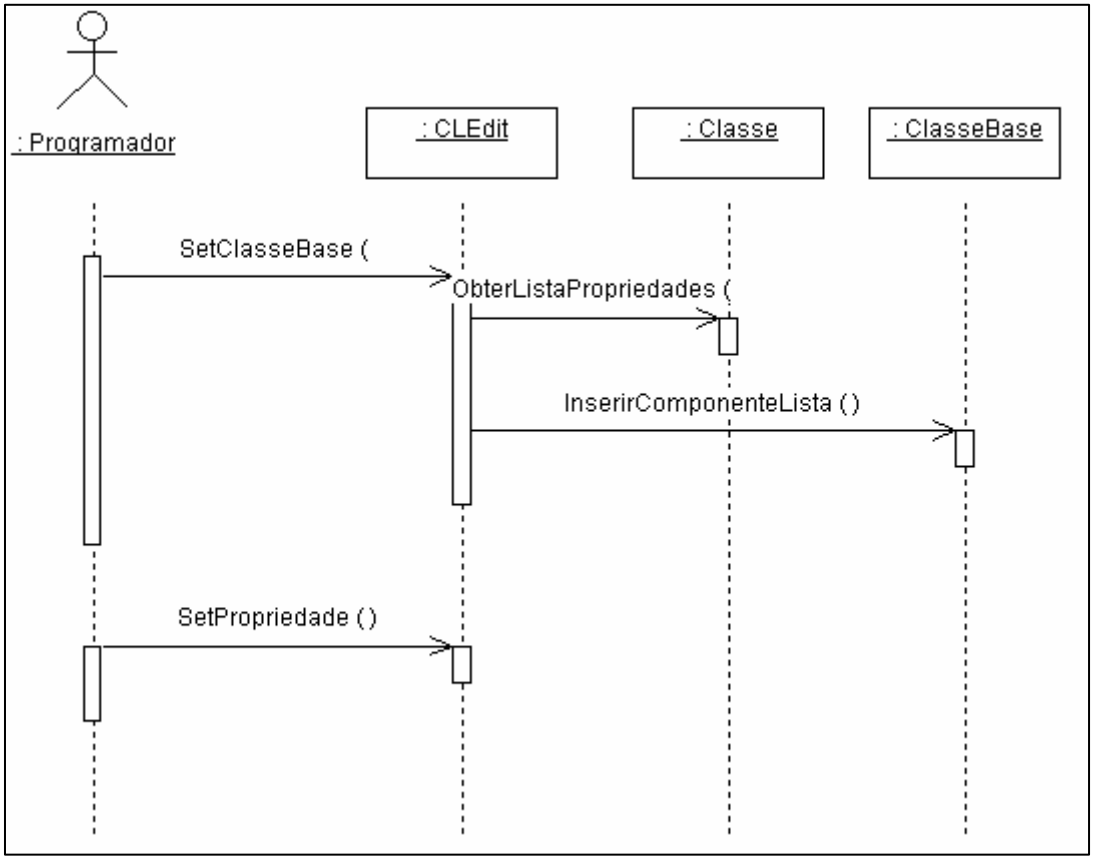
FIGURA 3-4: DIAGRAMA DE SEQÜÊNCIA DE INCLUSÃO DO COMPONENTE CLASSEBASE



Na inclusão do componente *ClasseBase*, o método *SetarGerenciadorClasse* permite ao programador vincular o componente *ClasseBase* ao componente *GerenciadorClasse*. O método *ObterListaClasses* busca do componente *GerenciadorClasse* a lista de classes carregadas anteriormente. O método *SetarClasse* possibilita ao programador a escolha de uma das classes anteriormente carregadas.

A figura 3-5 apresenta o diagrama de seqüência para a inclusão do componente *ClEdit.*

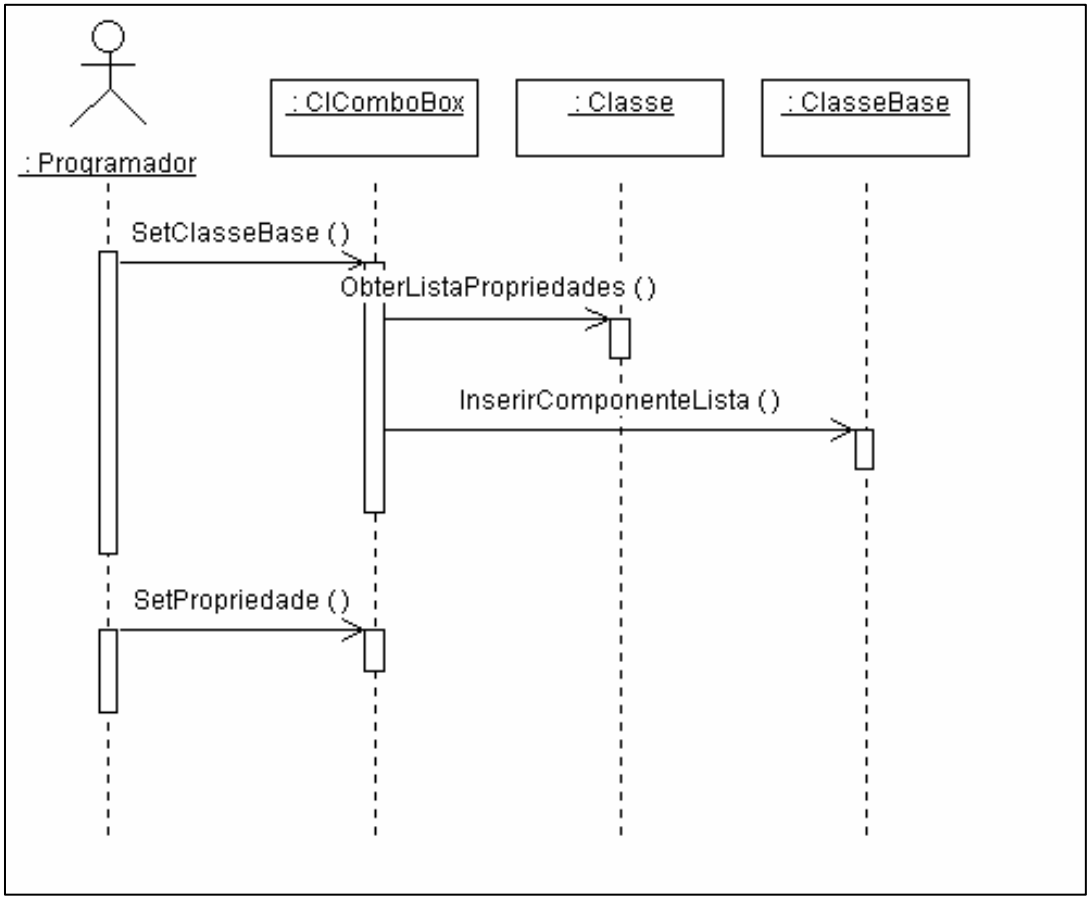
FIGURA 3-5: DIAGRAMA DE SEQÜÊNCIA DE INCLUSÃO DO COMPONENTE CLEDIT



Na inclusão do componente *ClEdit*, o método *SetClasseBase* permite ao programador vincular o componente *ClEdit* ao componente *ClasseBase*. O método *ObterListaPropriedades* busca da classe *Classe* a lista de propriedades carregadas anteriormente. O método *InserirComponenteLista* inclui o componente *ClEdit* na lista de componentes vinculados ao componente *ClasseBase*. O método *SetPropriedade* possibilita ao programador a escolha de uma das propriedades anteriormente carregadas.

A figura 3-6 apresenta o diagrama de seqüência para a inclusão do componente *ClComboBox*.

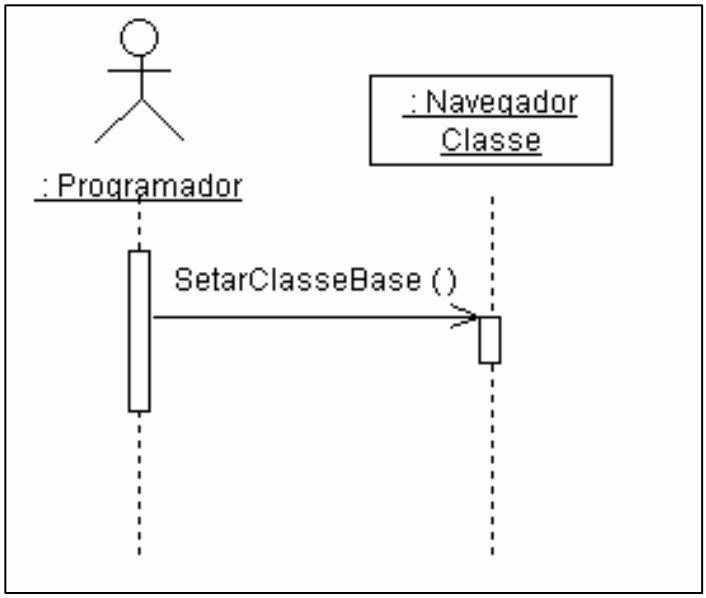
FIGURA 3-6: DIAGRAMA DE SEQÜÊNCIA DE INCLUSÃO DO COMPONENTE CLCOMBOBOX



Na inclusão do componente *ClComboBox*, o método *SetClasseBase* permite ao programador vincular o componente *ClComboBox* ao componente *ClasseBase*. O método *ObterListaPropriedades* busca da classe *Classe* a lista de propriedades carregadas anteriormente. O método *InserirComponenteLista* inclui o componente *ClComboBox* na lista de componentes vinculados ao componente *ClasseBase*. O método *SetPropriedade* possibilita ao programador a escolha de uma das propriedades anteriormente carregadas.

A figura 3-7 apresenta o diagrama de seqüência para a inclusão do componente *NavegadorClasse.*

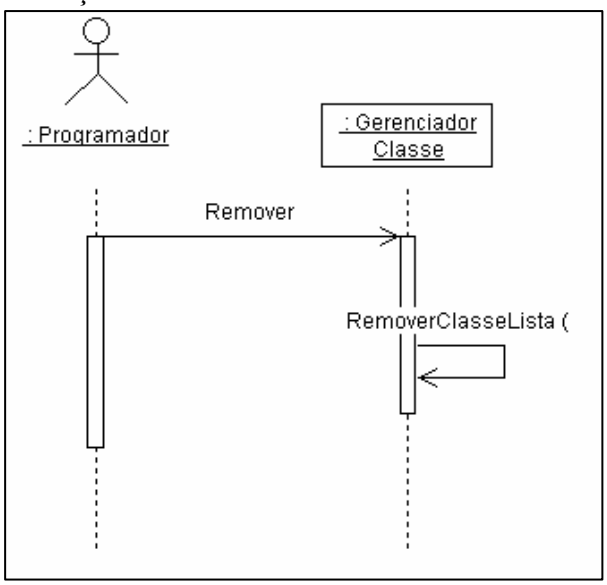
FIGURA 3-7: DIAGRAMA DE SEQÜÊNCIA DE INCLUSÃO DO COMPONENTE NAVEGADORCLASSE



Na inclusão do componente *NavegadorClasse*, o método *SetClasseBase* permite ao programador vincular o componente *NavegadorClasse* ao componente *ClasseBase*, permitindo assim, a chamada dos métodos para o controle de classes do componente *ClasseBase*.

A figura 3-8 apresenta o diagrama de seqüência para remoção do componente *GerenciadorClasse.*

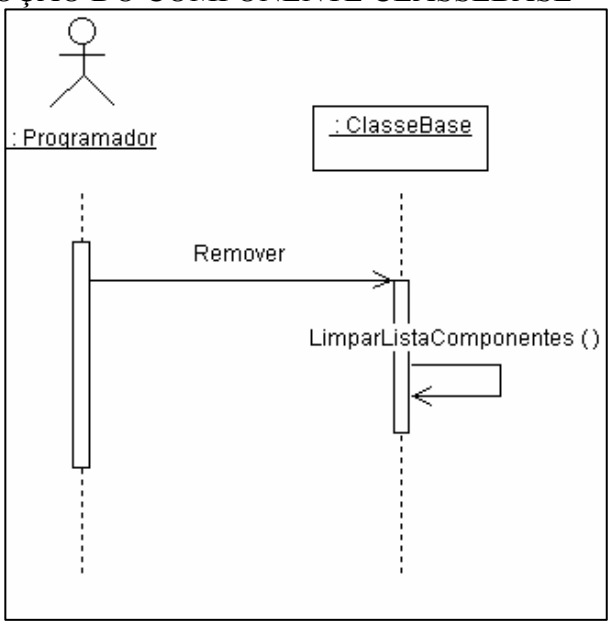
FIGURA 3-8: DIAGRAMA DE SEQÜÊNCIA DE REMOÇÃO DO COMPONENTE GERENCIADORCLASSE



Na exclusão do componente *GerenciadorClasse*, o método *RemoverClasseLista* remove as classes carregadas da lista de classes.

A figura 3-9 apresenta o diagrama de seqüência para remoção do componente *ClasseBase.*

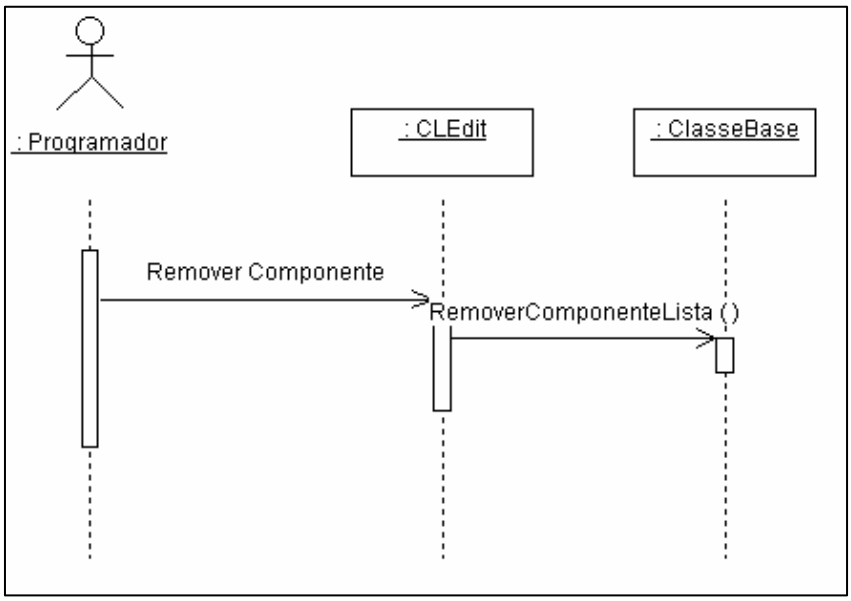
FIGURA 3-9: DIAGRAMA DE SEQÜÊNCIA DE REMOÇÃO DO COMPONENTE CLASSEBASE



Na exclusão do componente *ClasseBase*, o método *LimparListaComponentes* remove todos os componentes vinculados ao *ClasseBase*.

A figura 3-10 apresenta o diagrama de seqüência para remoção do componente *ClEdit.*

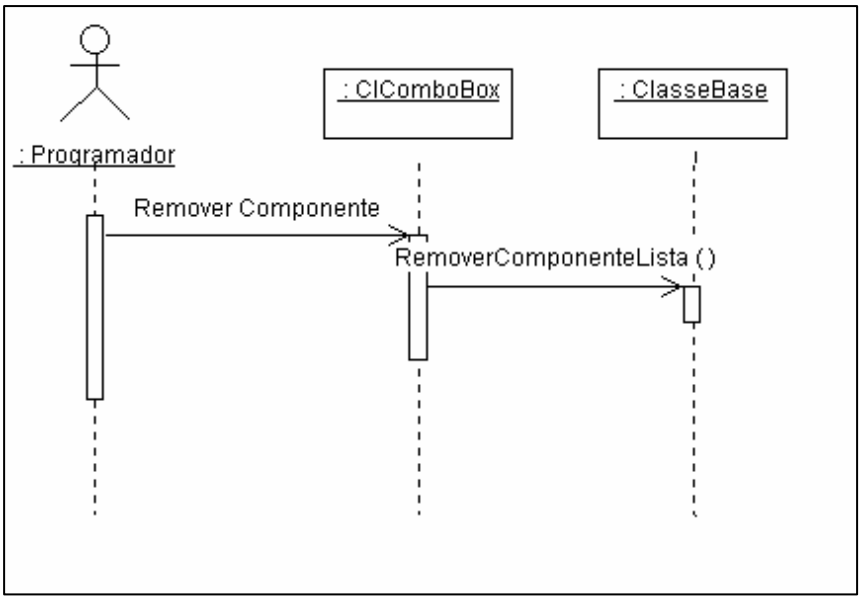
FIGURA 3-10: DIAGRAMA DE SEQÜÊNCIA DE REMOÇÃO DO COMPONENTE CLEDIT



Na exclusão do componente *ClEdit*, o método *RemoverComponenteLista* remove o *ClEdit* da lista de componentes do *ClasseBase*.

A figura 3-11 apresenta o diagrama de seqüência para remoção do componente *ClEdit.*

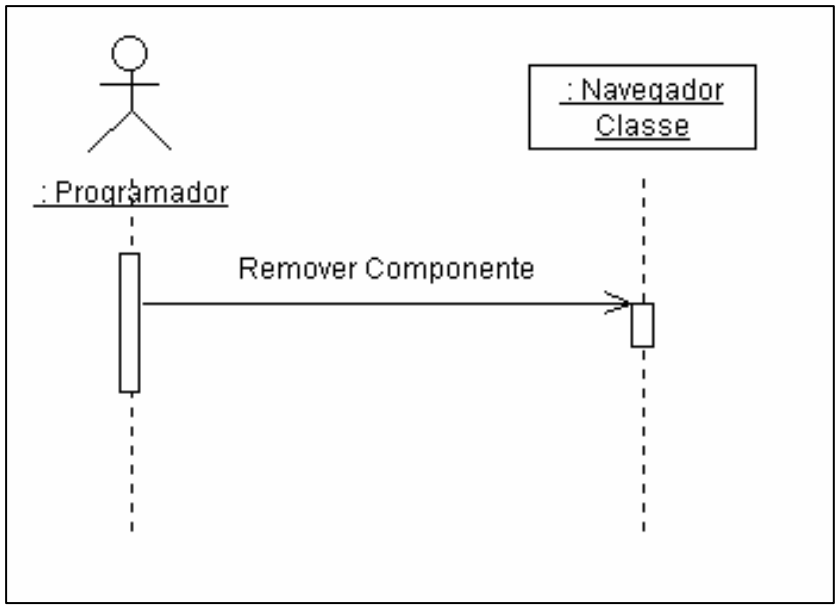
FIGURA 3-11: DIAGRAMA DE SEQÜÊNCIA DE REMOÇÃO DO COMPONENTE CLCOMBOBOX



Na exclusão do componente *ClComboBox*, o método *RemoverComponenteLista* remove o *ClComboBox* da lista de componentes do *ClasseBase*.

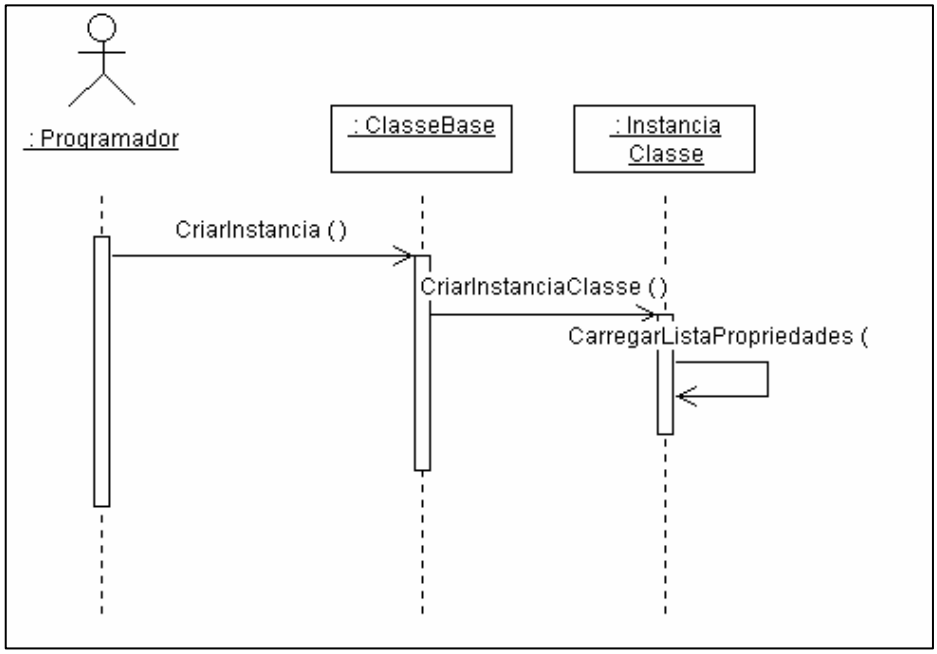
A figura 3-12 apresenta o diagrama de seqüência para remoção do componente *ClNavegadorClasse.*

FIGURA 3-12: DIAGRAMA DE SEQÜÊNCIA DE REMOÇÃO DO COMPONENTE CLNAVEGADORCLASSE



A figura 3-13 apresenta o diagrama de seqüência para a criação da instância.

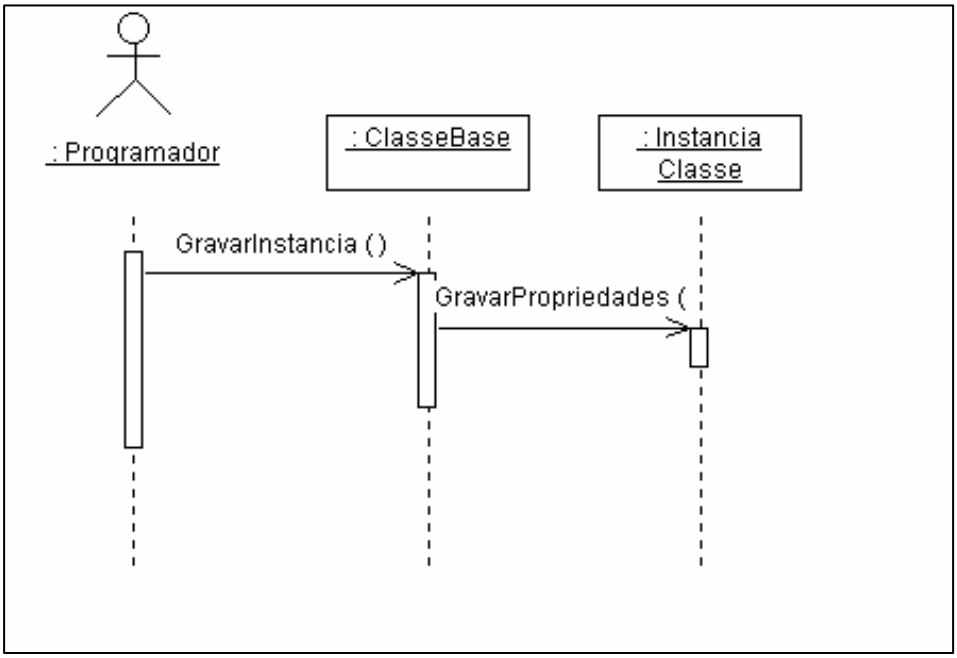
FIGURA 3-13: DIAGRAMA DE SEQÜÊNCIA DE CRIAÇÃO DA INSTÂNCIA



Na criação do registro o programador utiliza o método *CriarInstancia* que chama o método *CriarInstanciaClasse* criando uma instância da classe *InstanciaClasse*. O método *CarregarListaPropriedades* busca as propriedades da classe selecionada no componente *ClasseBase*.

A figura 3-14 apresenta o diagrama de seqüência para a gravação da instância.

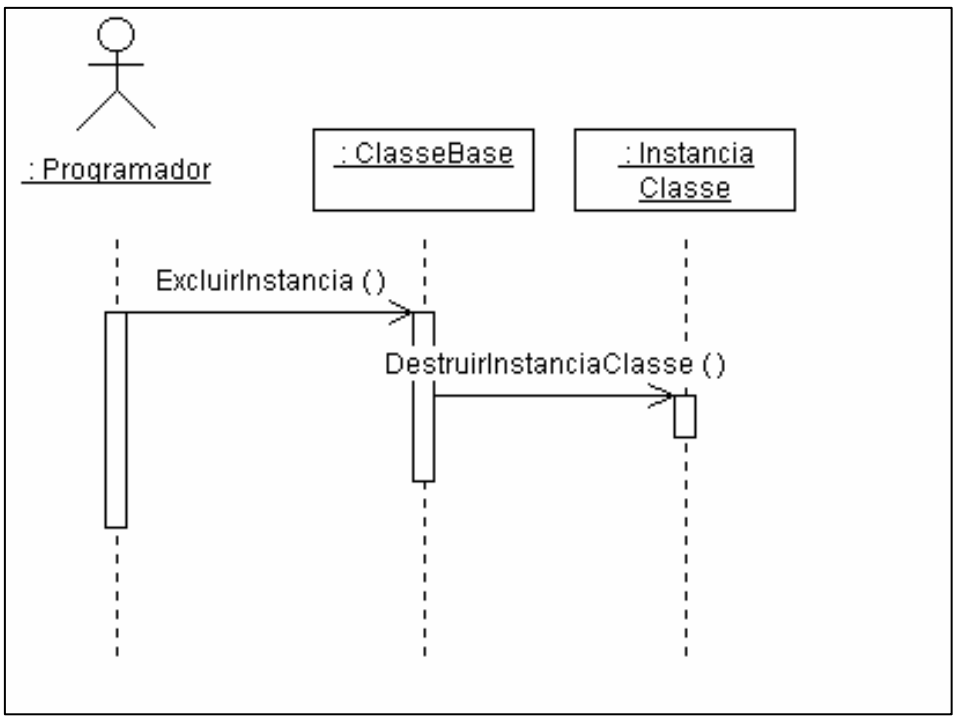
FIGURA 3-14: DIAGRAMA DE SEQÜÊNCIA DE GRAVAÇÃO DA INSTÂNCIA



Na gravação do registro o programador utiliza o método *GravarInstancia*. O método *GravarPropriedades* busca os valores das propriedades definidas pelo programador nos componentes *ClEdit* e *ClCombobox* para armazena-los em um arquivo.

A figura 3-15 apresenta o diagrama de seqüência para a exclusão da instância.

FIGURA 3-15: DIAGRAMA DE SEQÜÊNCIA DE EXCLUSÃO DA INSTÂNCIA



Na exclusão do registro o programador utiliza o método *ExcluirInstancia*. O método *DestruirInstanciaClasse* elimina a instância da classe *InstanciaClasse*.

3.3 IMPLEMENTAÇÃO

Considerações sobre as técnicas utilizadas para implementação do protótipo, bem como sua forma de apresentação, serão apresentados nesta seção.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O pacote de componentes foi implementado na linguagem Object Pascal no ambiente Delphi, versão 6.0, empregando os conceitos de orientação a objetos para desenvolver as classes responsáveis pela extração das classes e propriedades do código fonte.

O programador utiliza o componente *GerenciadorClasse* para varrer a *unit* definida pelo mesmo através do método *VarrerUnit*, buscando as classes e propriedades declaradas. O método *VarrerUnit* é apresentado no quadro 3-1.

QUADRO 3-1: MÉTODO INSERIRCLASE_LISTA

```
procedure TGerenciadorClasse.VarrerUnit;
var
  Linha_Pura, Inicio_Linha, Espaco_Branco : String;
  Arquivo, Novo_Arq  : TStringList;
  i, pos_chave_class, pos_chave_property,
  pos_chave_implementation : Integer;
  ClasseAtual : TClasse;
begin
if (FNomeUnit <> '') then
  begin
  try
  Inicio_Linha         := '#@#';
  Espaco_Branco        := '@@@';
  Novo_Arq             := TStringList.Create;
  Arquivo              := TStringList.Create;
  Arquivo.LoadFromFile(FNomeUnit);

  // Varre a Unit selecionada
  for i := 0 to Arquivo.Count - 1 do
     Novo_Arq.Add(Inicio_Linha + Substituir_String(Arquivo.Strings[i], ' ',Espaco_Branco));

  ClasseAtual       := Nil;

  for i := 0 to Novo_Arq.Count - 1 do
     begin
     Linha_Pura               := Substituir_String(Substituir_String(Novo_Arq.Strings[i],
                                 Inicio_Linha,''),Espaco_Branco,'');
     pos_chave_class          := pos('=CLASS', UpperCase(Substituir_String(Novo_Arq.Strings[i],
                                 Espaco_Branco,'')));
     pos_chave_implementation  := pos(Inicio_Linha + 'IMPLEMENTATION',UpperCase(Novo_Arq.Strings[i]));
     pos_chave_property        := pos(Inicio_Linha + 'PROPERTY',UpperCase(
                                 Substituir_String(Novo_Arq.Strings[i],Espaco_Branco,'')));

     if (pos_chave_class > 0) then // Se achar a palavra chave CLASS
        begin
        if (pos('=CLASS;', UpperCase(Substituir_String(Novo_Arq.Strings[i],Espaco_Branco,''))) < 1) and
           (pos('=CLASS(', UpperCase(Substituir_String(Novo_Arq.Strings[i],Espaco_Branco,''))) < 1) and
           (not (VerificaClasseLista(Obter_Nome_Classe(Linha_Pura)))) then
           ClasseAtual := InserirClasseLista(Linha_Pura); // Cria a instância da classe encontrada
        end;
     if (pos_chave_property > 0) and // Se achar a palavra chave PROPERTY
        (ClasseAtual <> Nil) then
        ClasseAtual.InserirPropriedadeLista(Linha_Pura); // Cria a instância da propriedade encontrada
     if (pos_chave_implementation > 0) then // Se achar a palavra chave IMPLEMENTATION para o processo
        break;
     end;
  finally
  Arquivo.Free;
  end;
end;
```

Este método varre a *Unit* selecionada. Ao encontrar a palavra chave *class*, a classe *Classe* é instanciada através do método *InserirClasseLista* armazenando as classes encontradas, e ao encontrar a palavra chave *property*, a classe *Propriedade* é instanciada através do método *InserirPropriedadeLista* armazenando as propriedades encontradas. Este método ignora os eventos, trabalhando somente com as propriedades da classe.

O método *InserirClasseLista* é apresentado no quadro 3-2.

QUADRO 3-2: MÉTODO INSERIRCLASELISTA

```
function TGerenciadorClasse.InserirClasseLista(Linha : String) : TClasse;
var AClasse : TClasse;
begin
  AClasse             := TClasse.CriarClasse; // Cria a intancia da classe Classe
  AClasse.NomeClasse  := Obter_Nome_Classe(Linha);
  FListaClasses.Add(AClasse); // Adiciona a instancia criada na lista de instâncias
  result              := AClasse;
end;
```

O método *InserirPropriedadeLista* é apresentado no quadro 3-3.

QUADRO 3-3: MÉTODO INSERIRPROPRIEDADELISTA

```
procedure TClasse.InserirPropriedadeLista(Nome_Propriedade : String);
var APropriedade : TPropriedade;
begin
  APropriedade                    := TPropriedade.CriarPropriedade; // Cria a intancia da classe
Propriedade
  APropriedade.NomeProp           := ObterNomePropriedade(Nome_Propriedade);
  APropriedade.Classe             := Self;
  FListaPropriedade.Add(APropriedade); //Adiciona a instancia criada na lista de instâncias
end;
```

O programador utiliza os métodos *CriarInstancia, GravarInstancia, ExcluirInstancia* e *GravarArquivo* do componente *ClasseBase* para manipular as instâncias da classe *InstanciaClasse* que é uma classe com a mesma lista de propriedades da classe definida pelo programador no componente *ClasseBase*.

O método *CriarInstancia* é apresentado no quadro 3-4.

QUADRO 3-4: MÉTODO CRIARINSTANCIA

```
procedure TClasseBase.CriarInstancia;
begin
  if (FStatusInstancia = [srNormal]) then
     begin
     FStatusInstancia     := [srInsercao];
     LimpaComponentes;
     FClasseAtual         := XGerenciadorClasse.ObterObjetoClasse(Classe);
     FInstanciaAtual      := TInstanciaClasse.CriarInstanciaClasse;
     // Busca a lista de propriedades da classe definida pelo programador
     FInstanciaAtual.CarregarListaPropriedades(FClasseAtual.ObterListaPropriedades);
     end;
end;
```

O método *CriarInstancia* cria uma instância da classe *InstanciaClasse*. A lista de propriedades da classe definida pelo programador é carregada através do método *CarregarListaPropriedades*.

O método *GravarInstancia* insere a instância criada pelo método *CriarInstancia* na lista de instâncias, esse método é apresentado no quadro 3-5.

QUADRO 3-5: MÉTODO GRAVARINSTANCIA

```
procedure TClasseBase.GravarInstancia;
begin
   if (FStatusInstancia = [srInsercao]) then
      begin
      FListaInstancia.Add(FInstanciaAtual);
      FStatusInstancia   := [srNormal];
      end;
end;
```

O método *ExcluirInstancia* exclui a instância criada pelo método *CriarInstancia* da lista de instâncias, posicionando a propriedade *InstanciaAtual* na última instância incluída anterior a essa. Esse método é apresentado no quadro 3-6.

QUADRO 3-6: MÉTODO EXCLUIRINSTANCIA

```
procedure TClasseBase.ExcluirInstancia;
var index : Integer;
begin
   if (FStatusInstancia = [srNormal]) then
      begin
      Index          := FListaInstancia.IndexOf(FInstanciaAtual);
      if (Index >= 0) then
         begin
         FListaInstancia.Delete(Index); // Exclui a instância selecionada da lista
         FInstanciaAtual.DestruirInstanciaClasse; // Destroi a instância selecionada
         if (Index > 0) then
            begin
            // Posiciona na última instância incluida
            if (TInstanciaClasse(FListaInstancia[Index - 1]) <> Nil) then
               FInstanciaAtual   := TInstanciaClasse(FListaInstancia[Index - 1]);
            end
         else
            FInstanciaAtual      := Nil;
         end;
      end;
end;
```

O método *GravarArquivo* varre as instâncias criadas e gera um arquivo texto com as instâncias gravadas. Esse método é apresentado no quadro 3-7.

QUADRO 3-7: MÉTODO GRAVARARQUIVO

```
procedure TClasseBase.GravarArquivo;
var ArquivoSaida : TStringList;
    ListaPropriedades : TList;
    i, j : Integer;
begin
ArquivoSaida    := TStringList.Create;
for i := 0 to FListaInstancia.Count - 1 do
    begin
    ArquivoSaida.Add('Instancia=' + IntToStr(i));
    ListaPropriedades := TInstanciaClasse(FListaInstancia[i]).ObterListaPropriedades;
    for j := 0 to ListaPropriedades.Count - 1 do
        begin
        ArquivoSaida.Add('      Propriedade=' + TPropriedade(ListaPropriedades[j]).NomeProp);
        ArquivoSaida.Add('      Valor=' + TPropriedade(ListaPropriedades[j]).ValorProp);
        end;
    ArquivoSaida.Add('------------------------');
    ArquivoSaida.Add('');
    end;
ArquivoSaida.SaveToFile(FgerenciadorClasse.ArquivoSaida);
ArquivoSaida.Free;
end;
```

Ainda no componente *ClasseBase*, o programador tem disponível os métodos *Primeiro*, *Próximo, Anterior* e *Ultimo* que fazem a navegação das instâncias criadas da classe *InstanciaClasse*, após mover a propriedade *InstanciaAtual*, o método *AtualizaComponentes*, atualiza os componentes visuais com os dados da instância atual. Estes métodos são apresentados no quadro 3-8.

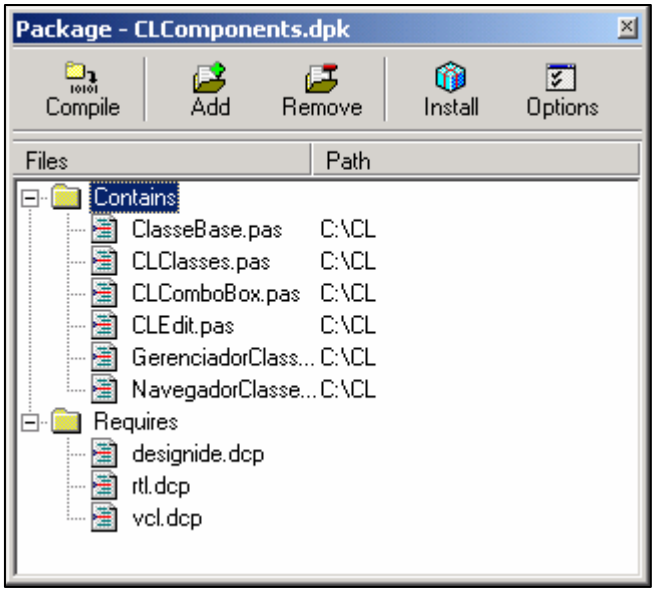
QUADRO 3-8: MÉTODO PRIMEIRO, PROXIMO, ANTERIOR E ULTIMO

```
procedure TClasseBase.Primeiro;
begin
   try
   // Posiciona na primeira instância criada da lista
   FInstanciaAtual   := TInstanciaClasse(FListaInstancia.First);
   except
   end;
   AtualizaComponentes;
end;

procedure TClasseBase.Proximo;
var PosicaoAtual : Integer;
begin
   PosicaoAtual      := FListaInstancia.IndexOf(FInstanciaAtual);
   try
   if (TInstanciaClasse(FListaInstancia[PosicaoAtual + 1]) <> Nil) then
      FInstanciaAtual   := TInstanciaClasse(FListaInstancia[PosicaoAtual + 1]);
   except
   end;
   AtualizaComponentes;
end;

procedure TClasseBase.Anterior;
var PosicaoAtual : Integer;
begin
   PosicaoAtual      := FListaInstancia.IndexOf(FInstanciaAtual);
   try
   if (TInstanciaClasse(FListaInstancia[PosicaoAtual - 1]) <> Nil) then
      FInstanciaAtual   := TInstanciaClasse(FListaInstancia[PosicaoAtual - 1]);
   except
   end;
   AtualizaComponentes;
end;

procedure TClasseBase.Ultimo;
begin
   try
   FInstanciaAtual   := TInstanciaClasse(FListaInstancia.Last);
   except
   end;
   AtualizaComponentes;
end;
```

A seguir serão apresentadas algumas telas do pacote de componentes na sua instalação.

A figura 3-16 mostra o pacote Cl Componentes criado para armazenar os componentes desenvolvidos para a biblioteca.

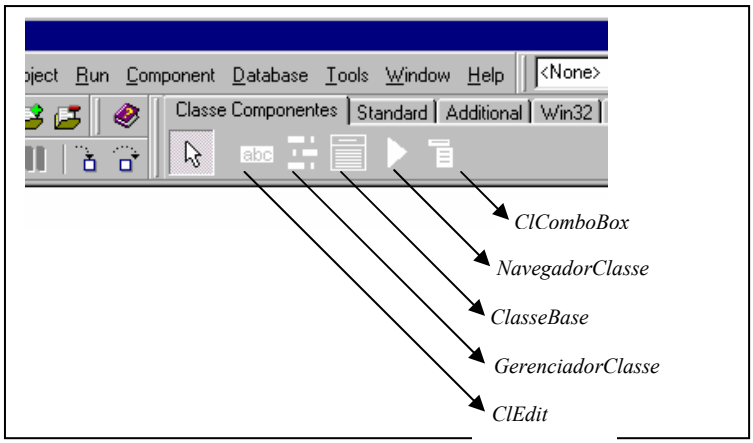
FIGURA 3-26: PACOTE CLCOMPONENTES



A seguir serão apresentadas algumas telas do pacote de componentes com suas respectivas propriedades, com o intuito de facilitar a compreensão.

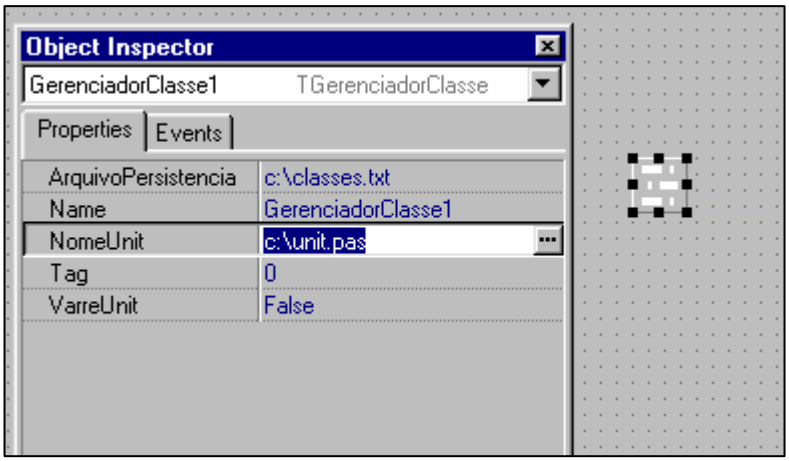
A figura 3-17 mostra a paleta de componentes do pacote no ambiente Delphi.

FIGURA 3-17: PALETA DO PACOTE DE COMPONENTES



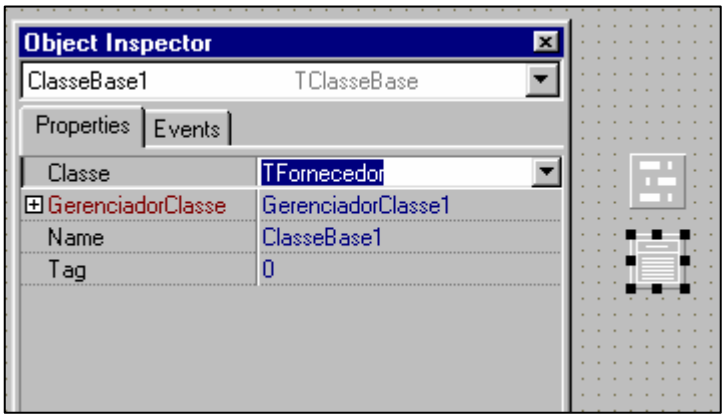
A figura 3-18 mostra o componente *GerenciadorClasse* com sua tela de propriedades.

FIGURA 3-18: COMPONENTE GERENCIADORCLASSE



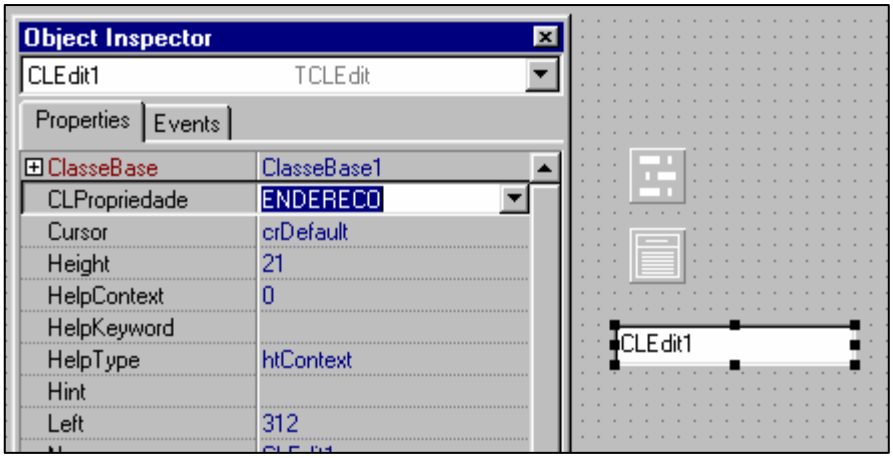
A figura 3-19 mostra o componente *ClasseBase* com sua tela de propriedades.

FIGURA 3-19: COMPONENTE CLASSEBASE



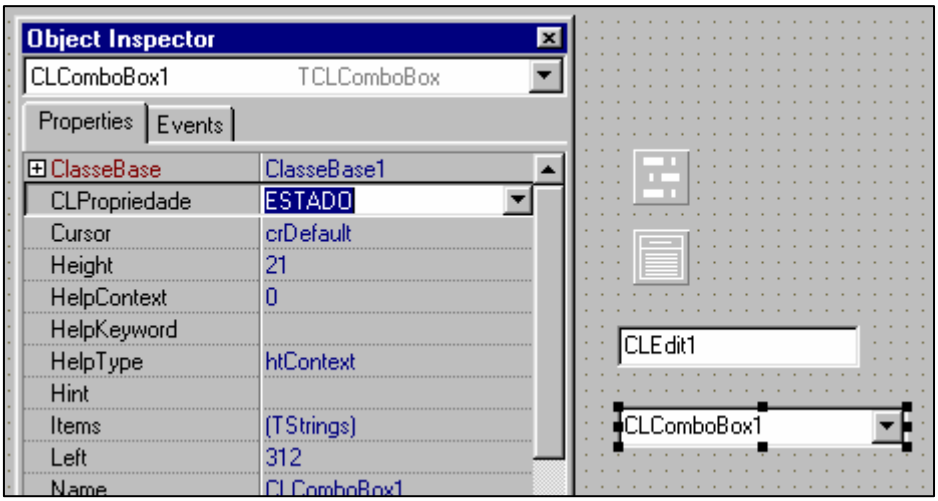
A figura 3-20 mostra o componente *ClEdit* com sua tela de propriedades.

FIGURA 3-20: COMPONENTE CLEDIT



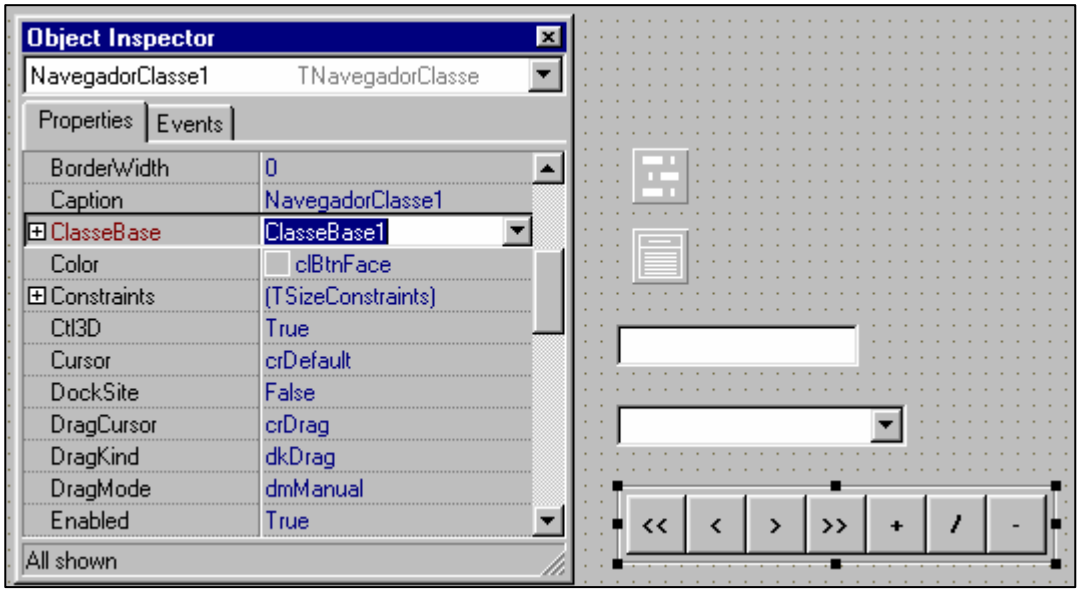
A figura 3-21 mostra o componente *ClComboBox* com sua tela de propriedades.

FIGURA 3-21: COMPONENTE CLCOMBOBOX



A figura 3-22 mostra o componente *NavegadorClasse* com sua tela de propriedades.

FIGURA 3-22: COMPONENTE CLCOMBOBOX



3.3.3 RESULTADOS E DISCUSSÃO

Para exemplificar a utilização do pacote de componentes, foi analisado o código fonte do projeto fictício de um sistema de cadastro de clientes e fornecedores. A figura 3-23 apresenta o seu diagrama de classes.

FIGURA 3-23: DIAGRAMA DE CLASSES DO SISTEMA DE CLIENTES E FORNECEDORES

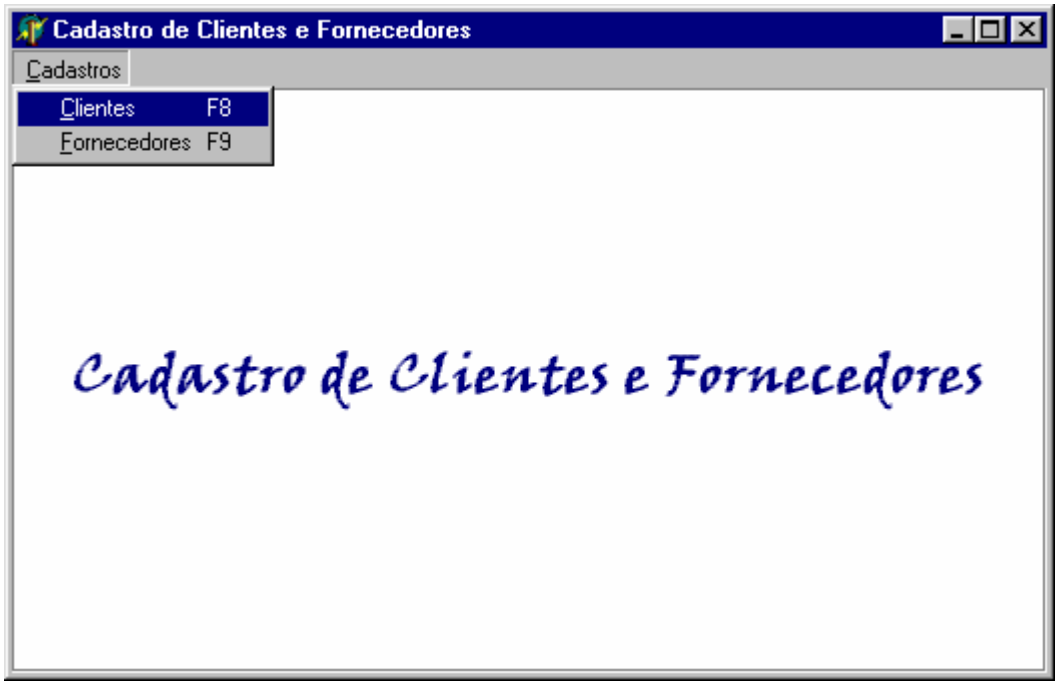
No quadro 3-9, apresentam-se as classes *Cliente* e *Fornecedor* que estão declaradas na *unit Uclasses.pas* do projeto.

QUADRO 3-9: MÉTODO PRIMEIRO, PROXIMO, ANTERIOR E ULTIMO

```
unit UClasses;

interface

type
  TCliente = class
  private
    FNome          : String;
    FEndereco      : String;
    FSexo          : String;
    FEstadoCivil   : String;
    FBairro        : String;
    FCidade        : String;
    FEstado        : String;
    FTelefone      : String;
  public
    property Nome : String read FNome write FNome;
    property Endereco : String read FEndereco write FEndereco;
    property Sexo : String read FSexo write FSexo;
    property EstadoCivil : String read FEstadoCivil write FEstadoCivil;
    property Bairro : String read FBairro write FBairro;
    property Cidade : String read FCidade write FCidade;
    property Estado : String read FEstado write FEstado;
    property Telefone : String read FTelefone write FTelefone;
  end;

  TFornecedor = class
  private
    FCGC           : String;
    FRazaoSocial   : String;
    FEndereco      : String;
    FBairro        : String;
    FCidade        : String;
    FEstado        : String;
    FTelefone      : String;
  public
    property CGC : String read FCGC write FCGC;
    property RazaoSocial : String read FRazaoSocial write FRazaoSocial;
    property Endereco : String read FEndereco write FEndereco;
    property Bairro : String read FBairro write FBairro;
    property Cidade : String read FCidade write FCidade;
    property Estado : String read FEstado write FEstado;
    property Telefone : String read FTelefone write FTelefone;
  end;

implementation

end.
```

A figura 3-24 apresenta a tela principal do sistema, onde se encontram os menus para a escolha do cadastro a ser feito.

FIGURA 3-24: TELA PRINCIPAL DO SISTEMA



A figura 3-25 apresenta a tela de cadastro de clientes em tempo de *design*. Nesta tela podem-se observar os componentes *GerenciadorClasse* e *ClasseBase*, no canto superior esquerdo, bem como o componente *NavegadorClasse* no campo inferior esquerdo, além dos componentes *ClEdit* e *CLComboBox* ao centro da tela.

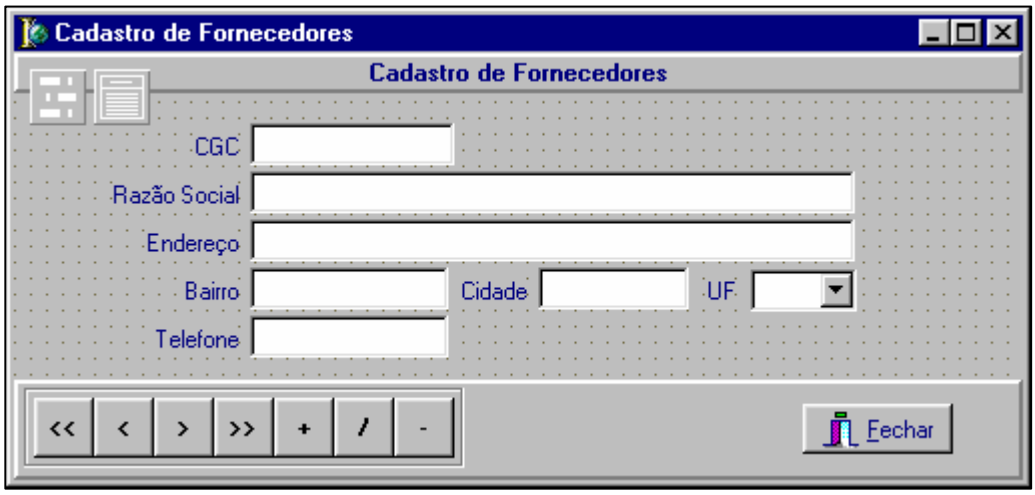
FIGURA 3-25: TELA DE CADASTRO DE CLIENTES EM TEMPO DE DESIGN

A figura 3-26 apresenta a tela de cadastro de clientes em tempo de execução, onde o componente *NavegadorClasse* chama os métodos *Primeiro, Anterior, Próximo, Ultimo, CriarInstancia, GravarInstancia* e *ExcluirInstancia* do componente *ClasseBase*.

FIGURA 3-26: TELA DE CADASTRO DE CLIENTES EM TEMPO DE EXECUÇÃO

A figura 3-27 apresenta a tela de cadastro de fornecedores em tempo de *design*. Nesta tela podem-se observar os componentes *GerenciadorClasse* e *ClasseBase*, no canto superior esquerdo, bem como o componente *NavegadorClasse* no campo inferior esquerdo, além dos componentes *ClEdit* e *CLComboBox* ao centro da tela.

FIGURA 3-27: TELA DE CADASTRO DE FORNECEDORES EM TEMPO DE DESIGN



A figura 3-28 apresenta a tela de cadastro de fornecedores em tempo de execução, onde o componente *NavegadorClasse* chama os métodos *Primeiro, Anterior, Próximo, Ultimo, CriarInstancia, GravarInstancia* e *ExcluirInstancia* do componente *ClasseBase*.

FIGURA 3-28: TELA DE CADASTRO DE FORNECEDORES EM TEMPO DE EXECUÇÃO

No quadro 3-10, apresenta-se o arquivo texto de persistência da classe *Cliente* gerado pelo componente *ClasseBase*. No componente *GerenciadorClasse*, há uma propriedade que identifica o nome do arquivo de saída para persistência, e o componente *ClasseBase* se encarrega de formatar os objetos da classe *InstanciaClasse* no formato de texto.

QUADRO 3-10: ARQUIVO DE PERSISTÊNCIA DA CLASSE CLIENTE

```
Instancia=0
      Propriedade=NOME
      Valor=Ricardo
      Propriedade=ENDERECO
      Valor=
      Propriedade=BAIRRO
      Valor=
      Propriedade=CIDADE
      Valor=Blumenau
-----------------------

Instancia=1
      Propriedade=NOME
      Valor=Jose
      Propriedade=ENDERECO
      Valor=
      Propriedade=BAIRRO
      Valor=
      Propriedade=CIDADE
      Valor=Indaial
-----------------------

Instancia=2
      Propriedade=NOME
      Valor=Manuel
      Propriedade=ENDERECO
      Valor=
      Propriedade=BAIRRO
      Valor=
      Propriedade=CIDADE
      Valor=Blumenau
-----------------------
```

4 CONCLUSÕES

O objetivo principal do trabalho, um pacote de componentes que auxilie de forma dirigida e consistente a criação de uma interface para o controle de objetos (instâncias), tomando como base as definições das classes foi atendido.

A programação orientada a objetos provê muitos benefícios como reutilização, facilidade em modificações, entre outros. Devido a estes fatores a programação orientada a objetos surge como uma possibilidade para a melhoria da qualidade e produtividade do software. Porém, não basta, apenas, pensar na qualidade do software sem pensar se ele está sendo desenvolvido, de uma forma legível e clara.

Com este pacote de componentes, o programador tem como vantagem o aumento da sua produtividade, já que os componentes desenvolvidos fazem por si só a interação entre as classes e propriedades definidas no projeto com as instâncias criadas, exigindo assim, pouca codificação no código fonte. Tornando assim menos trabalhoso o processo de desenvolvimento do software.

O pacote de componentes desenvolvido tem como característica facilitar ao programador do ambiente Delphi o controle dos objetos tomando como base as classes e propriedades declaradas no projeto, visando tornar o código fonte legível, melhorando assim, sua produtividade e qualidade de trabalho.

4.1 EXTENSÕES

Como extensões deste trabalho, sugerem-se:

a) Pesquisar um mecanismo que permita a instanciação das classes associadas aos componentes de forma que seja possível de trabalhar com os métodos e eventos das classes declaradas no projeto;

b) Ampliar a funcionalidade dos componentes, de modo que possibilite o uso de herança, agregação e associação de classes;

c) criar novos componentes visuais como, por exemplo, componentes equivalentes ao DBGrid, DBRadioGroup, DBCheckBox entre outros;

d) melhorar o mecanismo de persistência, utilizando, por exemplo, o padrão XML de forma que facilite a sua portabilidade com outros sistemas.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **Análise e projeto orientados a objeto.** Rio de Janeiro: Infobook, 1998.

BORATTI, Isaias C. **Programação orientada a objetos usando Delphi**. Florianópolis: Visual Books, 2002.

CANTÚ, Marco. **Delphi 6:** a bíblia**.** São Paulo: Makron Books, 2002.

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

GUERRA, Pedro Manuel; SANTOS, Silva Reis. **Reutilização de componentes de softwares com base em identificadores hierárquicos.** 2000. 271 f. Dissertação de doutorado (Doutor em Engenharia Informática e Computadores) Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa.

JONES, Meilir P. **Fundamentos do desenho orientado a objeto com UML.** São Paulo: Makron Books, 2001.

LEMAY, Laura; CADENHEAD, Rogers. **Aprenda em 21 dias java 2**. Rio de Janeiro: Campus, 2001.

LONGO, Maurício B.; SMITH, Ronaldo; POLISTCHUCK, Daniel. **Delphi 3 total.** Rio de Janeiro: Brasport, 1997.

MARTIN, James; ODELL, James J. **Análise e projeto orientados a objeto.** São Paulo: Makron Books, 1995.

McCLURE, Carma. **Software Reuse:** a standards-based guide. California: Computer Society, 2001.

PRESSMAN, Roger S. **Engenharia de software.** São Paulo: Makron Books, 1995.

RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William. **Modelagem e projetos baseados em objetos.** Rio de Janeiro: Campus, 1998.

SOMMERVILLE, Ian. **Engenharia de software**. São Paulo: Addison Wesley, 2003.

TEIXEIRA, Steve; PACHECO, Xavier. Borland Delphi 6 guia do desenvolvedor. Rio de Janeiro: Campus, 2002.