UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA DE ESTRUTURAS

"Automação de detalhamento de peças padronizadas em concreto armado via CAD e programação orientada a objetos"

Carla Soraia Leandro do Carmo

Dissertação apresentada ao Curso de Mestrado em Engenharia de Estruturas da Escola de Engenharia da Universidade Federal de Minas Gerais, como parte dos requisitos necessários à obtenção do título de "Mestre em Engenharia de Estruturas".

Comissão Examinadora:

_____

Prof. Dr. José Ricardo Queiroz Franco
DEES – UFMG – (Orientador)

_____

Prof. Dr. Ney Amorim Silva
DEES – UFMG

_____

Prof. Dr. Renato Cardoso Mesquita
CPDEE – UFMG

Belo Horizonte, 23 de fevereiro de 2001.

Ao meu filho Breno

Agradecimentos

Ao Professor José Ricardo Queiroz Franco por sua orientação e por ter acreditado em mim.

Aos meus queridos colegas da equipe CADTEC pela amizade, companheirismo, incentivo e apoio.

Aos funcionários e professores do Departamento de Estruturas da Escola de Engenharia da UFMG pela dedicação e atenção.

Ao meu pai Carlos, à minha mãe Vilma e à minha Tia Neuza por terem me ensinado a amar o conhecimento.

Ao meu marido Geraldo pela sua dedicação,estímulo e paciência.

Aos amigos do Programa de Pós Graduação em Engenharia de Estruturas da Escola de Engenharia da UFMG.

Resumo

Esta Dissertação de Mestrado consiste no desenvolvimento de um Sistema CAD para automatizar a execução do detalhamento de algumas peças componentes de uma estrutura de concreto convencional. Aplicando os conceitos da programação orientada a objetos à tecnologia CAD,o sistema foi desenvolvido para funcionar como um aplicativo dentro da plataforma gráfica AutoCAD. O escopo conceitual do seu desenvolvimento foi baseado na aplicação dos conceitos da engenharia de software, valiosos quando se quer desenvolver programas com qualidade. Por se tratar de um trabalho pioneiro na área de detalhamento, um grande esforço foi despendido na descrição das técnicas e recursos utilizados, com o objetivo de subsidiar da melhor maneira possível os futuros desenvolvimentos. Finalmente foram destacadas as contribuições deste sistema para o avanço da engenharia estrutural, suas limitações, sugestões e propostas para futuros trabalhos na mesma área.

Abstract

This M.Sc. dissertation consists of a CAD system development to automate the execution of the specification of some conventional concrete structure component pieces. Applying the concepts of Object-Oriented Programming to CAD technology, a system was developed as an application working under AutoCAD 2000 graphic platform. The conceptual mark of its development was based on the application of Software Engineering concepts, valuable when one wants to develop quality programs. As a pioneering work in specification area, a great effort was spent in the description of the used techniques and resources, with the objective of subsidizing, in the best possible way, futures developments. Finally it was discussed the outstanding contributions of this system for the progress of the Structural Engineering and its limitations and suggestions for futures works in the same area were proposed.

Sumário

Lista de Figuras

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Lista de Tabelas

Capítulo 2

Capítulo 5

Capítulo 6

Capítulo 1

INTRODUÇÃO

A Engenharia Estrutural e os Sistemas CAD

O Detalhamento Estrutural

Motivação e Objetivo

Resumo do Conteúdo de cada Capítulo

1.1 A Engenharia Estrutural e os Sistemas CAD

(Os sistemas CAD e a sua utilização como ferramenta no desenvolvimento de projetos estruturais)

A Engenharia Estrutural é uma especialidade da Engenharia Civil responsável por projetar a estrutura de sustentação das edificações residenciais e industriais e das chamadas obras de arte como pontes e viadutos.

A elaboração de um projeto estrutural envolve basicamente três etapas [**Figura 1-1**]. A interação entre estas etapas é necessária para que se chegue a uma solução ótima:

- Pré-processamento – compreende a entrada de dados que vai constituir o modelo da estrutura em análise, sua geometria, propriedades dos materiais, cargas, condições de apoio e vinculação dos elementos da estrutura.
- Processamento – consiste no cálculo dos esforços e deslocamentos na estrutura, produzidos pelos carregamentos possíveis.
- Pós-processamento – tratamento dos resultados obtidos da análise através do dimensionamento e detalhamento das peças que compõem a estrutura.

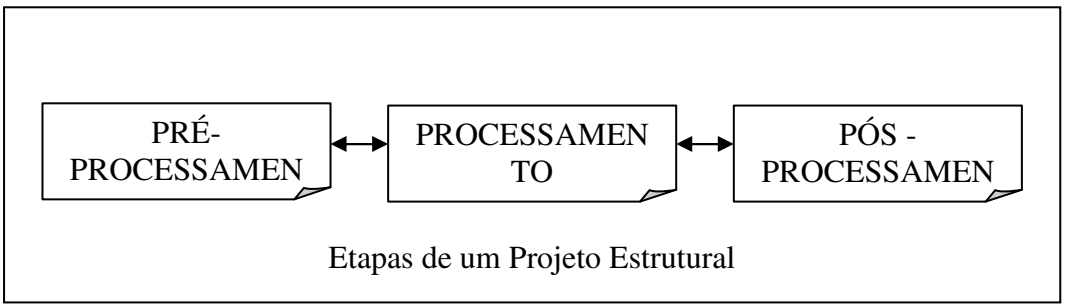


Figura 1-1

Os sistemas CAD (Computer Aided Design ou Projeto Assistido por Computador) foram desenvolvidos para auxiliar a projetar modelos manufaturáveis, utilizando ferramentas computacionais. No caso dos projetos estruturais, o modelo a ser construído será a estrutura e suas partes componentes.

No passado, além de raros, os sistemas existentes no mercado eram muito caros, exigiam equipamentos de grande porte e tinham que ser operados por especialistas em computação.

O surgimento das plataformas IBM PC/Intel, Apple Macintosh, Sun SparcStation e Silicon Graphics e de novos sistemas CAD de baixo custo , tais como o AutoCAD , Microstation e MiniCAD, promoveram a difusão dessa tecnologia tornando o computador uma ferramenta comum no escritório de projetos.

Hoje, os engenheiros projetistas já descobriram que a utilização de um sistema CAD é a única forma de manter a competitividade no mercado. Como ele possui mecanismos para agilizar os procedimentos, exige também uma maior capacidade para analisar e interpretar mais rapidamente eventuais erros de projeto.

A etapa de processamento foi a primeira a receber o auxílio da computação, permitindo automatizar os procedimentos de cálculo de esforços e deslocamentos. As etapas de pré e pós-processamento eram executadas manualmente a partir de arquivos e listagens cheias de números. Porém, esta automatização viabilizou a otimização dos resultados a níveis de complexidade proibitivos se realizados manualmente.

Com o desenvolvimento da tecnologia computacional e em particular das técnicas de processamento gráfico tornou-se viável representar visualmente no computador não só os resultados obtidos, mas também os próprios modelos analisados. A possibilidade de desenvolver aplicações utilizando a computação gráfica viabilizou a implementação de pré e pós-processadores gráficos poderosos, transformando a concepção estrutural em um processo dinâmico na busca da melhor alternativa.

Há no mercado sistemas para projetos de estrutura capazes de executar todas as etapas de cálculo, dimensionamento e análise das diversas partes da estrutura, além da geração de desenhos.

Alguns programas utilizados:

- CypeCAD (Multiplus Computação Gráfica) – Programa para Windows utilizado para cálculo de estruturas de concreto – análise estrutural, detalhamento, desenho, quantitativos e memorial automático de edifícios de concreto com até 5 pisos ou 30 pilares.
- Eberick e AltoQiFormas (AltoQi) – Eberick -Sistemas de cálculo estrutural para Windows com ambiente de CAD integrado. Permite lançar graficamente a estrutura sobre as plantas arquitetônicas, importadas via arquivo DXF. AltoQiFormas – sistemas para a geração de fôrma que lê os arquivos gerados pelo Eberick, montando a estrutura e gerando as fôrmas

para fabricação da estrutura com todos os detalhes, sem necessidade de interferência do usuário.

- Sistemas CAD/TQS (TQS Informática) – Sistemas de cálculo estrutural para o sistema operacional WINDOWS dotados de vários subsistemas para cálculo, dimensionamento, detalhamento e desenho das armaduras de Vigas, Pilares, Lajes e Fundações; cálculo, dimensionamento e detalhamento das estruturas de Madeira, Alvenaria Estrutural e Concreto Protendido além de módulos de otimização de processos de corte, dobra e transporte de barras de aço para construção civil e de corte, montagem e transporte de peças de madeira.
- Autocon Versão 2.1 - Programa de desenvolvido para a plataforma AutoCAD na linguagem AutoLISP que auxilia o engenheiro estrutural a desenvolver projetos de estruturas de concreto armado e protendido, especialmente desenhos de armaduras. É um aplicativo voltado para a elaboração de projetos de estruturas industriais, de obras de arte e de grandes obras.

1.2 Detalhamento Estrutural

(Histórico da concepção de projetos de detalhamento de estruturas utilizando ferramentas CAD (Projeto Assistido por Computador); funcionamento básico dos sistemas CAD; reflexões sobre a utilização de sistemas computacionais na execução do detalhamento de estruturas de concreto).

Como já dito, as primeiras ferramentas CAD aplicadas à engenharia estrutural contemplaram a etapa de processamento. Um programa passou a fornecer, de forma automatizada, os valores dos esforços e dos deslocamentos em cada peça. Na etapa de pós-processamento, estes dados eram então usados no seu dimensionamento e detalhamento. A demanda de tempo nesta etapa era considerável, transformando-a na mais onerosa do processo.

Com o avanço da Computação Gráfica, processadores gráficos poderosos foram desenvolvidos. A geração de imagens através de interfaces gráficas amigáveis contribuiu para consolidar a utilização de computadores nos escritórios de engenharia, em todas as etapas do projeto, aumentando a produtividade e a qualidade. Mas logo se notou que os computadores estavam se transformando em meras pranchetas

eletrônicas, principalmente na etapa do pós-processamento. Era necessário automatizar o processo de detalhamento de projetos para ganhar competitividade.

Realizar o detalhamento de uma estrutura de concreto não é uma tarefa trivial. É preciso especificar o diâmetro das armaduras necessário para combater cada um dos vários tipos de esforços a que estão sujeitos os componentes da estrutura, especificando a sua forma e tamanho. Com estas informações pode-se montar um modelo representativo da peça a ser confeccionada. Este modelo deverá seguir normas especiais de acordo com o tipo de estrutura projetada. Além disso, ele deve se adequar às necessidades construtivas e ao padrão de cada empresa.

Vários sistemas CAD foram desenvolvidos para projetar e detalhar estruturas em concreto armado, com características distintas e que podem ser esquematizadas como na **Figura 1-2**. No CASO 1, o sistema é composto de vários módulos que trocam informações entre si gerando o desenho das peças conforme parâmetros especificados em seus manuais de utilização. Exemplo: Sistemas CAD/TQS. No CASO 2, o sistema é composto por módulos separados formados por programas diferentes, cada um responsável por uma etapa do projeto.Exemplo: Eberick, AltoQiFormas e Autocon. A interação entre estes programas se faz através de um arquivo, que pode ser do tipo texto. Outra forma de interação pode ser uma entrada de dados. A separação dos módulos em programas distintos permite que o usuário tenha mais controle do processo proporcionando uma maior flexibilidade na adoção dos parâmetros de dimensionamento e detalhamento.

Ao longo destes últimos anos, o engenheiro de estruturas tem tentado se adaptar aos Sistemas CAD existentes no mercado. O ganho operacional com a utilização destes software é muito grande. Porém, a escolha de um programa deve sempre merecer uma atenção especial. Ele deve ser analisado sob o ponto de vista da sua confiabilidade, dos parâmetros adotados e da sua aceitação no mercado. É necessário compreender como são executadas todas as etapas do processo para que eventuais erros de projeto possam ser detectados. Os conhecimentos teóricos aliados à prática profissional são determinantes quando se quer obter resultados satisfatórios na utilização de programas CAD.

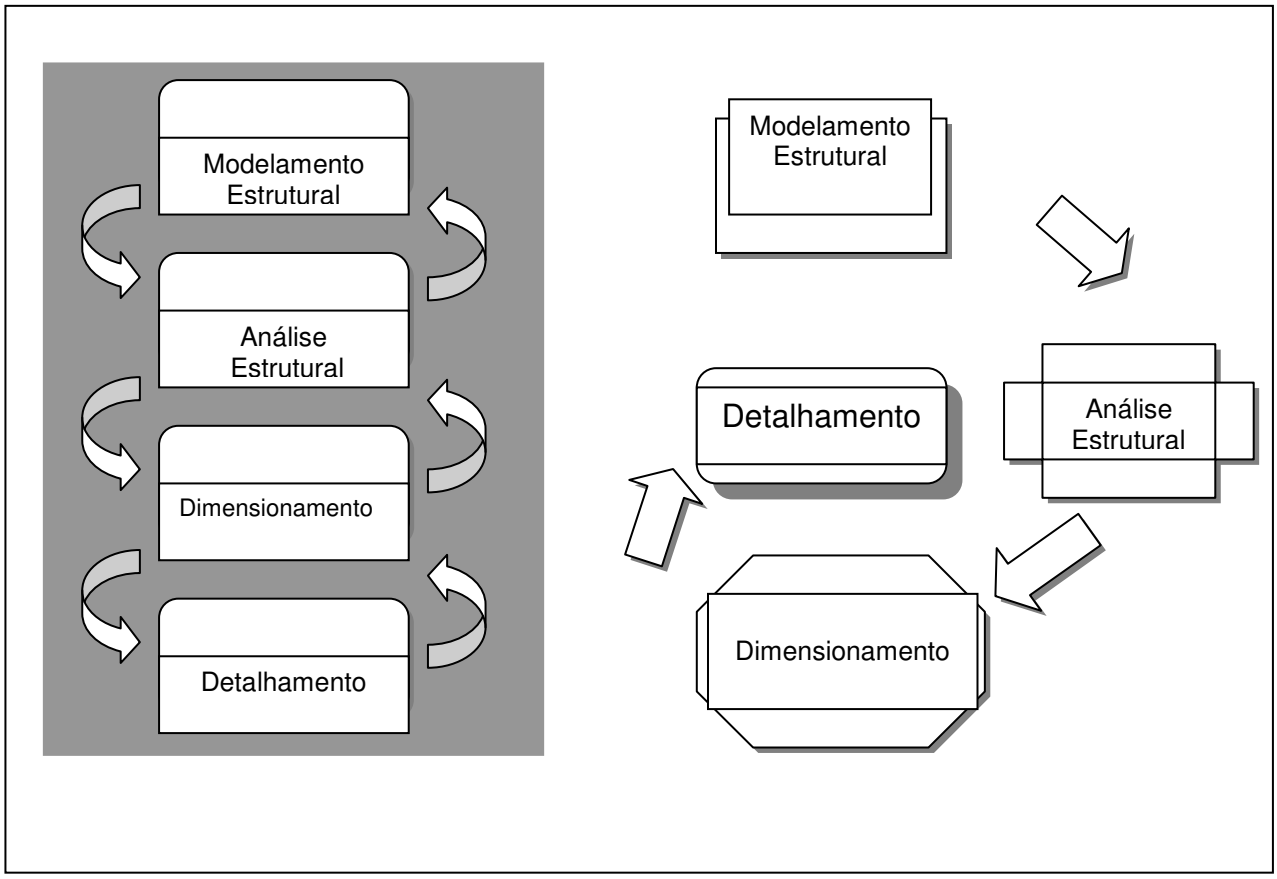


Figura 1-2

Devido a imensa quantidade de variantes encontradas em um desenho de detalhamento, muitas vezes, os software buscam minimizar a entrada de dados através de uma padronização de certos procedimentos. Isto parece aceitável, pois os próprios engenheiros calculistas o fazem quando detalham as peças manualmente. Como cada programa é configurado de forma a atender certos parâmetros de detalhamento, o desenho obtido muitas vezes não atende às necessidades individuais do usuário ou até mesmo não se adapta bem a certos tipos de modelos estruturais. Torna-se necessário então uma conferência minuciosa do projeto para fazer esta adaptação.

Portanto a tendência é adaptar ou produzir programas personalizados. A empresa define os requisitos e o sistema é adaptado para atender às suas necessidades. Esta opção tem se mostrado adequada embora o custo do desenvolvimento do software se torne às vezes impraticável.

Personalizar a custo baixo tem sido o maior desafio dos profissionais que desenvolvem tais sistemas.

1.3 Motivação e Objetivo

(Reflexões sobre a nova tecnologia, motivação e objetivos a serem alcançados com este trabalho).

Muitos desafios estão sendo impostos com o desenvolvimento de novas tecnologias. Um exemplo é o avanço da computação que se mostra presente em todas as áreas do conhecimento.

A busca incessante pela completa interação entre o homem e o computador tem contribuído para o desenvolvimento de programas que buscam se adaptar á necessidades particulares do usuário.

Em se tratando de sistemas CAD, destaca-se a plataforma gráfica AutoCAD. Ela já possui mecanismos para o desenvolvimento de aplicativos personalizados, utilizando a programação orientada a objetos. Este modelo de programação permite uma completa abstração da realidade do problema para o ambiente computacional, de forma clara e consistente.

Valendo-se desta poderosa plataforma gráfica e das facilidades da programação orientada a objeto, muitos desses aplicativos estão sendo desenvolvidos em todo o mundo, inclusive no Brasil. Neste caso, um exemplo importante é o trabalho desenvolvido na empresa CODEME, fruto de um projeto que contou com uma parceria entre a UFMG e a empresa. Os resultados alcançados foram bastante significativos, pois possibilitou posteriormente a integração da tecnologia CAD à tecnologia CAM (Computer Aided Manufacturing ou Manufatura Assistida por Computador). Desenvolvendo sistemas de automação que dominam o processo de produção desde a concepção do projeto até a fabricação das peças componentes, esta empresa conseguiu chegar a um bom nível de automação de seu processo.

Seguindo esta linha de pesquisa com resultados já comprovados, o objetivo deste trabalho é dar início ao desenvolvimento de um aplicativo para a plataforma AutoCAD, que execute de forma automatizada, o detalhamento da armadura de peças de concreto armado. Tal programa deverá alcançar os seguintes objetivos:

- Ser capaz de produzir o desenho da forma e o desenho da armadura de algumas peças em concreto armado.
- Possuir fornecer uma lista de ferros individual para cada peça e outra para o conjunto de peças detalhadas naquele desenho.

- Possuir uma interface com usuário fácil e intuitiva. Para alcançar esta interação será utilizado o ambiente Windows.
- Possuir dispositivos de edição da fôrma e da armadura da peça, permitindo a sua modificação de forma simples, consistente e organizada.
- Possuir um banco de dados, capaz de armazenar os dados de fôrma confiável.
- Ser arquitetado de forma a facilitar a sua adaptação às necessidades particulares de cada usuário.
- Deverá contemplar, de forma consciente, as normas na ABNT que tratam do detalhamento em estruturas de concreto armado.
- Garantir, através de uma boa documentação do seu código, as informações necessárias para facilitar a manutenção e a evolução do programa.

O produto deste trabalho, em sua versão inicial, e será o resultado da aplicação dos conceitos da programação orientada a objetos à tecnologia CAD, hoje, uma ferramenta fundamental na Engenharia Estrutural. Por se tratar de um trabalho pioneiro na área de detalhamento de concreto armado aplicando tais conceitos, um grande esforço foi despendido na descrição das técnicas e recursos utilizados, com o objetivo de subsidiar da melhor maneira possível os futuros desenvolvimentos.

1.4 Resumo do Conteúdo de cada Capítulo

Este trabalho encontra-se dividido em oito capítulos.

Capítulo 1 - Introdução:

Visão geral da aplicação da computação na Engenharia Estrutural e no Detalhamento Estrutural; introdução dos conceitos básicos necessários à compreensão do tema; motivação e objetivos deste trabalho.

Capítulo 2 - Ferramentas necessárias para o desenvolvimento do programa:

Descrição das características da programação orientada a objetos; apresentação da plataforma gráfica AutoCAD; definição do ambiente de desenvolvimento e das bibliotecas C++ a serem utilizadas na confecção do aplicativo; descrição das etapas de desenvolvimento de um programa utilizando a metodologia da

Engenharia de Software; contribuições do CADTEC para o desenvolvimento deste projeto.

Capítulo 3 - Planejamento do desenvolvimento do aplicativo:

Levantamento dos requisitos básicos para desenvolver um programa de detalhamento de estruturas de concreto; abstração dos elementos componentes do universo estudado e definição das classes de objetos do aplicativo a ser desenvolvimento; elaboração de um fluxograma, mostrando as etapas de desenvolvimento.

Capítulo 4 – Descrição dos Módulos do Aplicativo ArmaCAD:

Descrição detalhada de cada uma das classes que compõem o mapa de classes do aplicativo ArmaCAD , seus atributos e funções membro.

Capítulo 5 – Descrição da Interface com o usuário do Aplicativo ArmaCAD:

Descrição das classes derivadas de MFC responsáveis pela captação dos dados necessários para a construção das representações gráficas das peças construídas pelas classes do ArmaCAD.

Capítulo 6 – Conclusões:

Contribuições deste trabalho no desenvolvimento de Sistemas CAD para automatizar a execução do detalhamento de peça de estrutura em concreto armado. Limitações encontradas no aplicativo desenvolvido. Sugestões de trabalhos futuros que podem aumentar a sua aplicabilidade e funcionalidade.

Apêndice I - A Integração entre o Visual C++ e as Bibliotecas ObjectARX e MFC :

Instruções de como criar um projeto no Visual C++ 6.00 que possa utilizar a biblioteca em C++ ObjectARX; instruções de como formatar o Visual C++ para utilizar a Biblioteca em C++ MFC (Microsoft Fundation Class) em um projeto que utiliza o ObjectARX.

Apêndice I I - Manual de Referência do Aplicativo ArmaCAD Versão 1.0:

Descrição de como acessar e utilizar os comandos do aplicativo ArmaCAD.

Capítulo 2

FERRAMENTAS NECESSÁRIAS PARA O DESENVOLVIMENTO DO PROGRAMA

A Programação Orientada a Objetos- POO

A Plataforma Gráfica AutoCAD

O Visual C++ e as Bibliotecas ObjectARX e MFC

A Engenharia de Software

O CADTEC

2.1 A Programação Orientada a Objetos - POO

(Breve histórico do desenvolvimento das linguagens de programação, comparação entre a POO (Programação Orientada a Objeto) e a Programação Estruturada ;conceitos, características e vantagens da POO)

A metodologia para criar programas computacionais começou a se desenvolver na década de quarenta, porém nesta época não existiam linguagens de programação. Os procedimentos eram implementados diretamente ao nível de máquina. O surgimento das linguagens montadoras no início da década de cinqüenta estimulou o desenvolvimento de ferramentas que ajudassem na tarefa de programar.

Até o final da década de 60 a linguagem de programação passou por três etapas de desenvolvimento distintas. O objetivo era aprimorar os conceitos básicos de programação das expressões aritméticas, comandos, matrizes, listas, pilhas e procedimentos. As chamadas Linguagens de Primeira Geração FORTRAN I e ALGOL 58 precederam às Linguagens de Segunda Geração FORTRAN II, ALGOL 60, COBOL 61 e Lisp, surgindo posteriormente as Linguagens de Terceira Geração PL/I, ALGOL 68 e PASCAL. Também nesta terceira fase surge a linguagem *Simula*, a "mãe" das linguagens orientadas a objeto [Mon94].

Em 1968 foi estabelecida a premissa básica da Programação Estruturada: Todo programa pode ser escrito pela combinação de comandos primitivos ou subprogramas, independente do seu nível de complexidade. Para viabilizar a sua construção, já estavam disponíveis três estruturas de controle: comandos complexos executados seqüencialmente, comandos condicionais (if - then - else) e comandos de repetição ou laço(for, while e repeat ... until). Durante muitos anos, inúmeros sistemas foram desenvolvidos seguindo este paradigma. A decomposição do sistema em subprogramas força o programador a fixar a atenção muito mais nos procedimentos do que nos dados. Este é o ponto de desequilíbrio e uma das principais fontes de críticas à Programação Estruturada [Mon94].

A exigência crescente dos usuários tem forçado a indústria de software a acelerar o ritmo de produtividade e melhorar a qualidade de seus software. Assim, mais uma vez, o profissional da informática se vê diante de uma nova tecnologia e da decisão de mudança de paradigma. Se o aumento de produtividade era uma

necessidade crucial, como seria possível efetivá-lo? Indubitavelmente, através do reaproveitamento de esforços.

Historicamente, o amadurecimento dos princípios para alcançar este objetivo, começou na década de 60 com a Linguagem *Simula*, depois na década de 70 com a Linguagem *Smalltalk* e na década de 80 com a Linguagem *C++*.

Curiosamente, foi a popularização de uma linguagem não orientada a objetos, a linguagem C, que ajudou no desenvolvimento do C++. O C, firmou-se no mercado por possuir duas características fundamentais: um grande poder de programação e uma boa portabilidade devido à existência de vários compiladores para diferentes plataformas.

Durante a primeira metade dos anos 80 nos laboratórios da AT&T, desenvolvia-se o C++ , com o objetivo de integrar os poderes da Programação Orientada a Objeto presentes na *Simula* a mais popular linguagem de programação de sistemas, o C. Assim, os criadores do C++, intencionalmente, preservaram a compatibilidade com as linhas de código C, escritas ao longo de quase uma década [Mon94].

Atualmente, o poder do C++ como linguagem de implementação de programas orientados a objetos pode ser medido através dos investimentos feitos por empresas importantes como a Borland e a Microsoft que, no sentido de dominar este amplo mercado, criaram os seus ambientes de programação em PC compatíveis com esta linguagem.

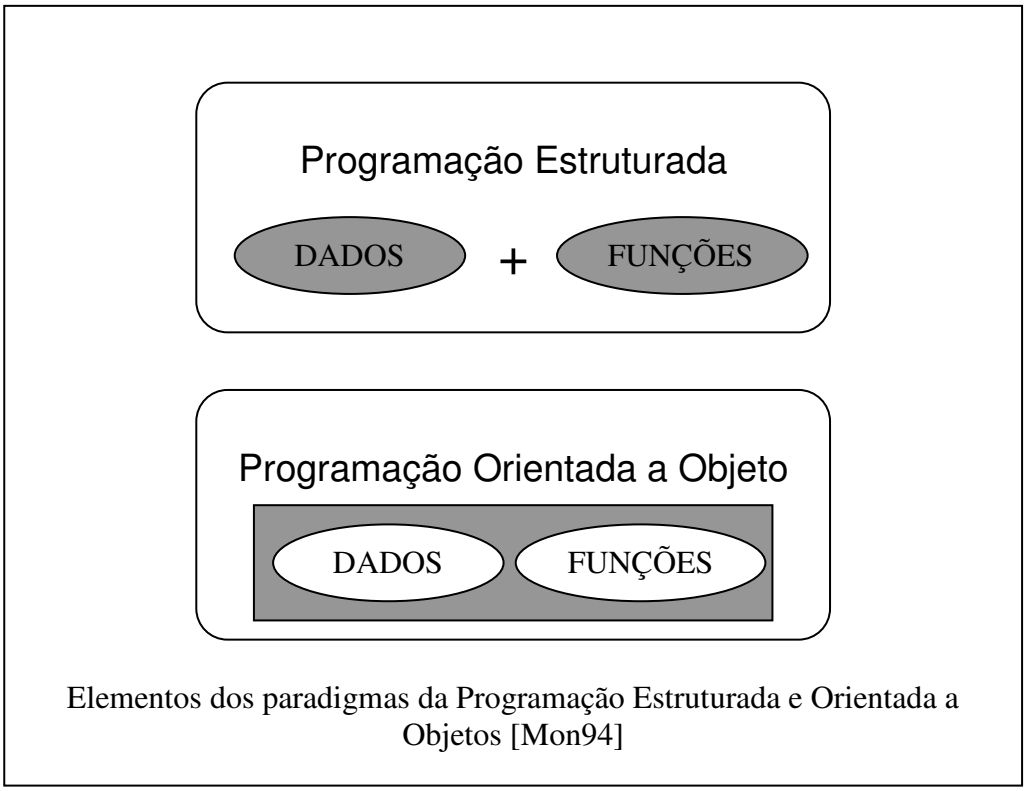
Traçando um paralelo entre o enfoque tradicional da Análise Estruturada e o enfoque da Orientação a Objetos percebe-se que a principal diferença entre eles é a forma pela qual dados e procedimentos se intercomunicam.

Nos Sistemas Estruturados, a implementação é feita através de subprogramas que se comunicam através da passagem de dados. Portanto, eles são estruturados na forma de procedimentos que têm como tarefa a manipulação dos dados.

Na Análise Orientada a Objetos, dados e procedimentos estão agrupados formando as classes de objetos. Tais classes descrevem um dado do tipo abstrato com sua implementação parcial ou total. Um dado do tipo abstrato representa um conjunto de objetos definidos por uma lista de operações aplicáveis a estes objetos e seus dados.[**Figura 2-1**].

Qual seria a vantagem da programação orientada a objeto em relação a estruturada? Por se tratar de uma programação modular baseada em classes, a POO

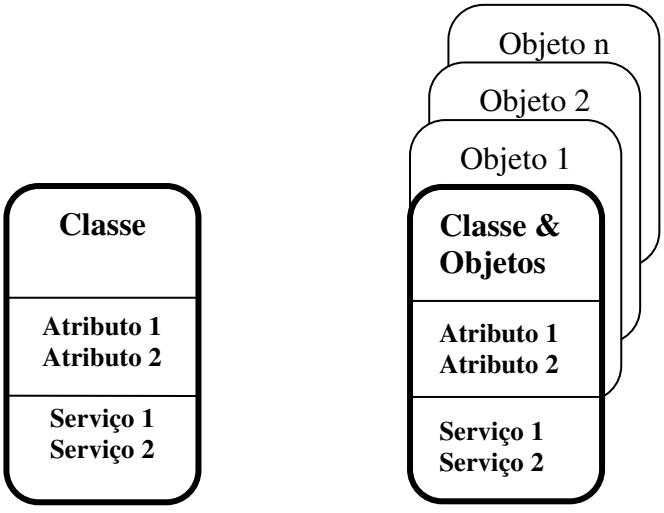
possui um alto índice de reutilização de código e maior facilidade de manutenção do sistema. O reaproveitamento favorece o desenvolvimento de programas mais robustos e eficientes. Assim, por exemplo, códigos referentes à interface, podem ser aproveitados em diversos programas, sem dificuldades na alteração.



Elementos dos paradigmas da Programação Estruturada e Orientada a Objetos [Mon94]

Figura 2-1

Um sistema orientado a objetos deve possuir características e mecanismos que o definam como tal. Também possui uma notação específica para representar estes diversos mecanismos. Várias notações estão disponíveis, mas a adotada neste trabalho foi a de Booch [Boo92a]e[Boo92b] para a definição e ilustração dos conceitos e Coad e Yourdon[Coa92] para notação gráfica. [**Figura 2-2**]. A seguir, estão fundamentados os mecanismos sobre os quais um sistema orientado a objetos se baseia com suas respectivas notações gráficas.

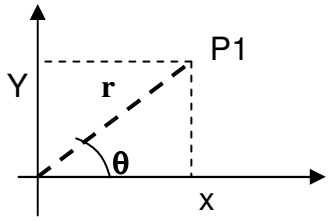


Notação gráfica de Classe Abstrata, Classe&Objeto, atributos e serviços [Coa96]

Figura 2-2

- **Objeto ou Instância**: Um objeto é uma instância de uma classe. Tal instância é uma estrutura de dados construída de acordo com parâmetros pré-definidos. Por exemplo, um objeto da classe PONTO é uma estrutura de dados que possui duas variáveis X e Y definidas nesta classe. A um objeto estarão sempre associadas o seu estado, comportamento e identidade. O estado do objeto é definido pelas propriedades que ele possui e pelos valores que elas estão assumindo. Por exemplo, o fato de P1, objeto da classe PONTO ter as coordenadas x = 2.5 e y = 1.0 definirá o seu estado. O comportamento é definido pela forma com que ele age e reage em termos de mudança de seu estado e seu relacionamento com os demais objetos do sistema. No caso do exemplo acima, o comportamento do objeto com os dados x e y pode ser alterado quando alguma operação de mudança for executada. A identidade é a propriedade da ocorrência única, ou seja, é a propriedade pela qual um objeto se distingue dos demais. Assim o ponto P1 é único no sistema [Mey97].
- **Classe**: Uma classe é uma abstração de dados, provido de atributos e métodos que resumem as características comuns de vários objetos. A classe PONTO representa um conjunto de pontos em coordenadas cartesianas no plano XY. Para estas classes, usa-se o termo Classe&Objeto Porém, existem algumas que não possuem nenhum objeto associado a elas. Neste caso, elas são conhecidas por Classes Abstratas. Por exemplo, se a classe PONTO tem como atributos, além das coordenadas cartesianas x e

y, as coordenadas polares r e θ, existirão duas formas diferentes de definir o objeto P1. Para tanto, podemos criar a Classe PONTO com métodos puramente virtuais, que serão implementados de formas diferentes nas classes PONTOCARTESIANO e PONTOPOLAR. Para representá-lo na forma cartesiana, basta fornecer os valores de x e y. Na forma polar é necessário realizar operações matemáticas com x e y para computar r e θ. A construção da Classe Abstrata PONTO, que não representa nenhum objeto ponto, possibilitou a criação da Classe&Objeto PONTOCARTESIANO, que representa um conjunto de pontos nas coordenadas cartesianas, e da Classe&Objeto PONTOPOLAR, representada por um conjunto de pontos nas coordenadas polares. O método que implementará a representação gráfica do objeto ponto é apenas uma das funções membro contidas na classe abstrata PONTO. [Mey97].[**Figura 2-3**]



Um ponto e suas coordenadas
[Mey97]

Figura 2-3

- **Membros de Classe**: São os Métodos e Atributos constituintes de uma Classe Abstrata ou de uma Classe&Objeto. Um atributo ou dado é uma informação definida para um objeto de classe. Ele tem a função de adicionar detalhes ao modelo que está sendo construído. Estes atributos deverão ser manipulados exclusivamente por serviços ou métodos associados à classe a que pertence ou por outras classes a ela associadas. Existem dois tipos de atributos em um sistema orientado a objetos: os atributos de objetos e os atributos de classe. Novamente para a classe PONTO, os atributos de objeto já foram definidos. Um exemplo de

atributo de classe poderia ser uma variável que armazena a quantidade de objetos desta classe, já construídos. Métodos ou serviços são operações efetuadas pelos objetos. Estes procedimentos são implementados no nível do objeto ao qual se relacionam. São três os tipos fundamentais de serviço em um programa orientado a objetos: Serviços de Ocorrência ou "Serviços Implícitos" (criar, alterar, deletar, acessar ou selecionar instâncias), Serviços de Cálculo (manipulação matemática dos atributos) e Serviços de Monitoração (manipulação de entradas e saídas externas ao sistema ou controle e aquisição de dados).

- **Herança**: Relação entre classes, de modo a permitir o compartilhamento de atributos e serviços semelhantes. Com a herança pode-se definir novas classes por extensão, especialização ou combinação de outras já definidas. Assim, uma classe mais especializada, por exemplo, a classe PONTOPOLAR, herda as propriedades da classe mais geral que é a classe PONTO. Quando uma classe-derivada possui características de mais de uma classe-base, tem-se o que se chama Herança Múltipla. Isto é o que realmente acontece em sistemas complexos. A herança é uma técnica chave quando se quer desenvolver sistemas reutilizáveis e adaptáveis [Mey97].

Algumas características são obrigatórias a um modelo baseado no paradigma da orientação a objetos, enquanto outras são desejáveis, porém não obrigatórias. Entre as essenciais estão:

- **Abstração**: Abstrair consiste em separar mentalmente elementos relevantes de um todo. O produto final de um programa representa um modelo de algum aspecto do mundo real. Através da abstração pode-se mapear as entidades do mundo real em objetos do programa. Na programação orientada a objetos isto acontece através do conceito de classes e objetos. A decisão pelo conjunto de abstrações correto é o principal fator de distinção entre a boa e a má análise orientada a objetos [Mon94].
- **Encapsulamento**: Este é o nome dado à propriedade de se implementar dados e procedimentos correlacionados em uma mesma entidade ou objeto. O objetivo é desenvolver sistemas que possam ser reutilizados sem depender da sua implementação interna. Assim as entidades usuárias de

bibliotecas de classe dependerão apenas da interface de acesso aos membros das classes que integram esta biblioteca, não importando como tais classes foram implementadas.

- **Polimorfismo**: É a habilidade de uma entidade de se adaptar a diferentes tipos de dados [Mey97]. O sistema deve ser capaz de discernir, dentre os métodos homônimos, aquele que deve ser executado. Isto se dá através da identificação do objeto receptor. Devido à existência de ligações dinâmicas, a ligação objeto-função só se dará em tempo de execução. O polimorfismo é um dos responsáveis pela facilidade de extensão de um programa orientado a objeto.
- **Modularidade**: Os sistemas complexos são formados por conjuntos de módulos independentes e a cada um deles está associada uma abstração. Cada módulo possui independência de funcionamento e pode ser compilado separadamente. Isto representa a otimização do processo de recompilação quando há alguma modificação nos módulos componentes. Outra característica da modularidade é a colocação em arquivos separados, a declaração e a implementação do código das classes. Além disso, esta característica permite que os programadores possam trabalhar em equipe, cada um trabalhando um módulo diferente [Mey97].

Estes conceitos e características facilitarão o desenvolvimento de um software orientado a objetos.

2.2 A Plataforma Gráfica AutoCAD

(Descrição das características do software AutoCAD , destacando as ferramentas disponíveis para utilização, costumização e aprendizado deste programa)

O Software AutoCAD é um ambiente de costumização, banco de dados e um conjunto de ferramentas para projetar em 2D e 3D. Ele é um sistema automatizado de CAD, líder no mercado, que tem se adaptado muito bem às necessidades essenciais encontradas ao desenvolver projetos estruturais. É executado em PC baseado na plataforma Windows e é aperfeiçoado para a Internet.

Autodesk foi a primeira a definir um CAD para categoria PC, quando introduziu o AutoCAD baseado em DOS em 1982. Desde então, várias versões foram

lançadas no mercado. Hoje o AutoCAD 2000i, possui recursos de comunicação via Web, inimagináveis a dezoito anos atrás. Através do refinamento do código, buscando manter-se sempre alinhado com as novas tecnologias, o programa adotou a programação orientada a objetos, a tecnologia COM (Microsoft Component Object Model), o núcleo ACIS de modelagem, e melhorias tecnológicas como Heidi e ISM.

Esta nova versão apresenta várias inovações tais como o ambiente de projeto múltiplo (MDE - Multiple Design Environment), que permite a abertura de vários desenhos em uma única seção e a cópia de um arquivo para outro. Visando aumentar as possibilidades de costumização e expansão, o AutoCAD 2000 soma às implementações em LISP, Visual Basic, ActiveX, a tecnologia ObjectARX para facilitar a personalização dos sistemas.

Tentando ser uma referência quando se fala em utilizar um programa CAD, a Autodesk investe em treinamento. Através do e-Learning, os usuários podem participar de cursos online, salas de aula virtuais, cursos personalizados para atender às necessidades de cada empresa, testes de avaliação para usuários e empregados. O material de ajuda que acompanha os seus software contém Livros e CD ROMs que facilitam a consulta e ajudam a aumentar os conhecimentos sobre o produto.

Para aqueles que se interessam em desenvolver aplicativos para a plataforma AutoCAD, estão disponíveis grupos de discussão online que permitem a troca de idéias.

A Autodesk promove também, a criação de Centros de Treinamento Autorizados junto a instituições educacionais e provedores de treinamento privados reconhecidos, que possam oferecer treinamento seguindo os padrões rigorosos estabelecidos por ela.

Visando oferecer produtos individualizados e soluções integradas, várias aplicações foram desenvolvidas para a Plataforma AutoCAD:

- AutoCAD Architectural Desktop - aperfeiçoado para a arquitetura.
- AutoCAD LT - solução em 2D que oferece facilidade de uso.
- Actrix Technical - ferramenta 2D de diagramação para criação de desenhos dinâmicos, planos de instalações, esquemas técnicos, tudo em um ambiente de desenho automatizado.
- AutoSketch - Versão barata e fácil de usar que fornece ferramentas para usuários de CAD que precisam criar desenhos conceituais e esboços.

- QuickCAD - programa CAD rápido, fácil de usar e disponível para usuários experientes e inexperientes.
- AutoCAD Express Tools - conjunto poderoso de rotinas criadas para economizar horas de desenvolvimento.
- WHIP! Viewer - Ajuda na visualização e impressão de desenhos CAD.
- Autodesk Symbols - bibliotecas que contêm mais de 8.000 símbolos arquitetônicos, mecânicos, e eletrônicos.
- Autodesk CAD Overlay - software que facilita a vetorização, convertendo os desenhos no papel para imagens digitalizadas no computador manipulando imagens raster.
- Autodesk AEC Object Enabler - possibilita a exibição e edição de objetos criados no AutoCAD sem precisar da aplicação Pai (AutoCAD). Livre.

O conhecimento dos comandos nativos, bem como o funcionamento geral do programa AutoCAD é imprescindível para o desenvolvimento do aplicativo proposto.

2.3 O Visual C++ e as Bibliotecas ObjectARX e MFC

(A escolha do ambiente de desenvolvimento Visual C++; visão geral da biblioteca ObjectARX , suas classes, componentes, banco de dados; visão geral da Biblioteca MFC e sua integração com o AutoCAD.)

Neste projeto, o ambiente de desenvolvimento adotado foi o Visual C++ 6.0. Esta é a única alternativa predefinida pela Autodesk que, em parceria com a Microsoft, possibilita a criação de aplicativos para a Plataforma AutoCAD 2000, utilizando a linguagem C++ e a Programação Orientada a Objeto.

Através do Visual C++ 6.0, pode-se desenvolver um aplicativo completo com interfaces visuais e barras de ferramentas usando a Biblioteca ObjectARX e a Biblioteca MFC (Microsoft Foundation Class).

A biblioteca ObjectARX é constituída por uma interface de programação usada para costumizar e estender o AutoCAD.

Com o ObjectARX é possível executar as seguintes tarefas:

- Acessar o banco de dados do AutoCAD.

- Interagir com o editor do AutoCAD.
- Criar interfaces usando o MFC (Microsoft Foundation Classes).
- Suportar a interface de múltiplos documentos - MDI(Multiple Document Interface).
- Criar classes costumizadas com características herdadas das classes do ObjectARX.
- Construir aplicações complexas usando as várias ferramentas disponíveis.
- Interagir com outros ambientes de programação tais como Visual LISP, ActiveX e COM ou com a Internet através da associação de URLs a entidades possibilitando carregar e salvar arquivos de desenho na Web.

A biblioteca ObjectARX está dividida em classes separadas pela sua funcionalidade [Aut99].(**Tabela 2-1**)

Grupo de Classes	**Função**
AcRx	Fazem a ligação de uma aplicação com o AutoCAD e registram e identificam através das ligações dinâmicas as classes chamadas, em tempo de execução.
AcEd	Contem o registro dos comandos e notificações de eventos do AutoCAD.
AcDb	É o banco de dados do AutoCAD.
AcGi	Possibilita a visualização gráfica das entidades do AutoCAD
AcGe	Contém as classes utilitárias com funções matemáticas, da geometria e álgebra linear.

Tabela 2-1 – Classes da Biblioteca ObjectARX

A família de classes AcRx, possui um sistema capaz de registrar e identificar uma outra classe em tempo de execução. Portanto, ela trabalha com ligações dinâmicas ao determinar a qual classe pertence o objeto que está sendo criado, através da identificação de sua herança. O ObjectARX possui um conjunto de macros que auxilia na criação de classes derivadas de AcRxObject, a classe base da família AcRx.

Dentre as bibliotecas definidas, destaca-se também o conjunto AcDb. É ele que compõe o banco de dados do AutoCAD. O banco de dados armazena todas as

informações dos objetos, aqui chamados de entidades, que fazem parte de um desenho, que tenham representação gráfica ou não. Exemplos: layer, tipo de linha, estilo de texto, as coordenadas dos pontos em um desenho.

O banco de dados possui os seguintes elementos [Aut99]:

- Um conjunto de nove tabelas de símbolos
  1. Block table (AcDbBlockTable)
  2. Dimension style table (AcDbDimStyleTable)
  3. Layer Table (AcDbLayerTable)
  4. Linetype table (AcDbLinetypeTable)
  5. Registered applications table (AcDbRegAppTable)
  6. Text style table (AcDbTextStyleTable)
  7. User Coordinate System table (AcDbUCSTable)
  8. Viewport table (AcDbViewportTable)
  9. View table(AcDbViewTable)
- Um dicionário (*named object dictionary*)
- Um conjunto fixo de variáveis pré-definidas.

Objetos armazenados no banco de dados que não possuem representação gráfica, herdarão as características da classe AcDbObject. Já os que possuem representação gráfica, serão derivados de AcDbEntity. [**Figura 2-4**]

Para criar uma classe derivada de AcDbObject deve-se sobrecarregar algumas funções. Elas são as capazes de armazenar os dados quando se deseja salvar um desenho ou fazer a sua leitura no AutoCAD. São elas: [Aut99]

- Virtual Acad::ErrorStatus dwgInFields(AcDbDwgFiler* filer).
- Virtual Acad::ErrorStatus dwgOutFields(AcDbDwgFiler* filer).
- Virtual Acad::ErrorStatus dxfInFields(AcDbDxfFiler* filer).
- Virtual Acad::ErrorStatus dxfOutFields(AcDbDxfFiler* filer).

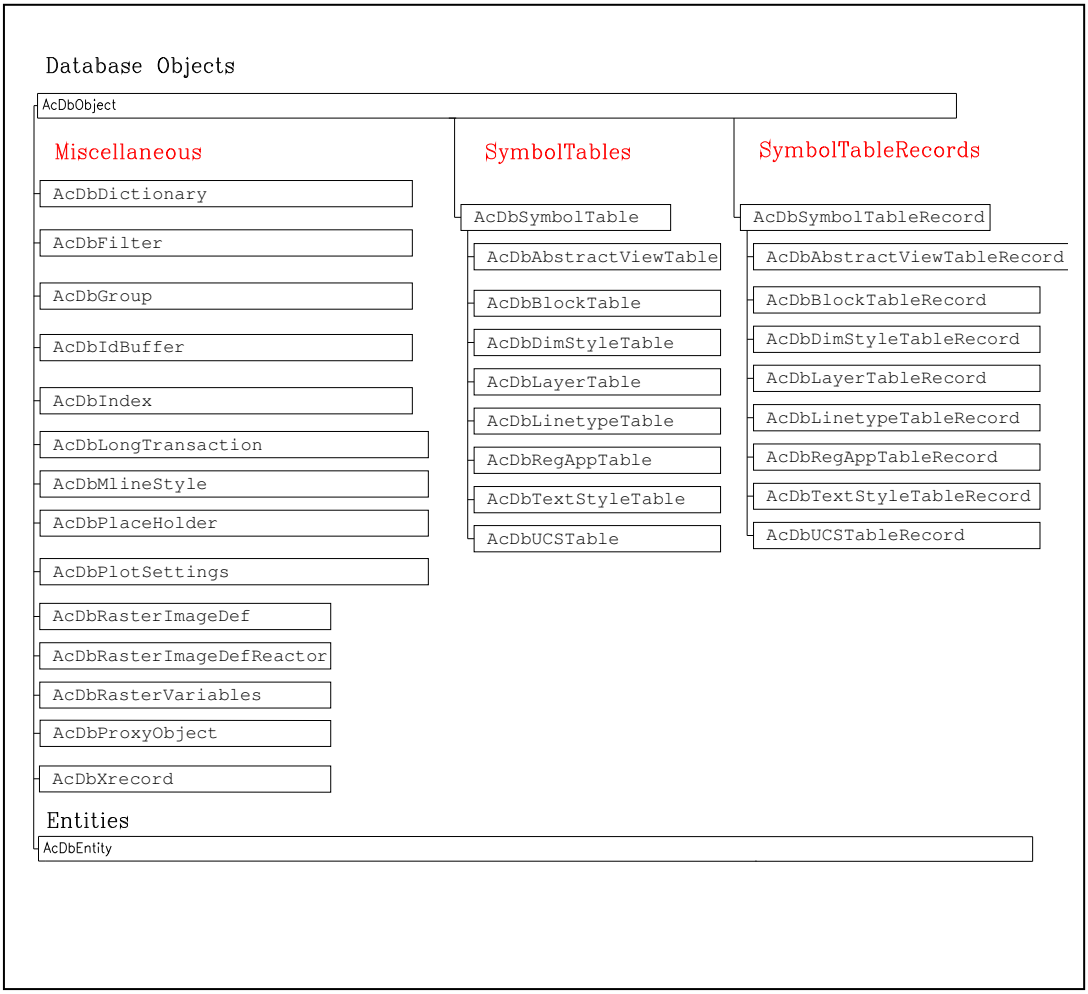


Figura 2-4

AcDbEntity é a classe base de todos os objetos que têm representação gráfica. Ela é derivada de AcDbObject, portanto herda todas as suas características e ainda possui várias outras necessárias a visualização da entidade na tela. Para criar um objeto costumizado , isto é , um objeto novo, diferente daqueles nativos do AutoCAD, é necessário seguir alguns passos [Aut99]:

1. Derivar a classe costumizada de AcDbEntity. Automaticamente, esta classe será derivada de AcDbObject.
2. Sobrecarregar todas as funções da classe AcDbObject, que irão armazenar os dados que não possuem representação gráfica..

3. Sobrecarregar todas as funções da classe AcDbEntity, que fornecerão os dados necessários para se obter a representação gráfica do objeto pretendida.
4. Sobrecarregar as outras funções, que irão dar suporte a funcionalidade do aplicativo.

Uma classe derivada de AcDbEntity, deverá, obrigatoriamente, sobrecarregar a função *worldDraw* descrita abaixo. Ela possui como atributo um ponteiro para a classe AcGiWorldDraw. Esta classe possui um conjunto de características geométricas que pertencem às entidades primitivas do AutoCAD. Através da função membro *geometry()* as primitivas, Circle, Circular Arc, Polyline, Polygon, Mesh, Shell, Text, Xline, e Ray, podem ser desenhadas em coordenadas globais, bastando fornecer os parâmetros exigidos pela função, de acordo com a primitiva a ser desenhada. Por exemplo: para desenhar uma polyline deve-se fornecer as coordenadas de pelo menos dois pontos. Um conjunto de primitivas formará então o desenho do objeto pretendido, de forma costumizada.

- Virtual Adesk::Boolean worldDraw(AcGiWorldDraw *pWd).

A Biblioteca MFC (Microsoft Foundation Class) é um conjunto de classes de objetos que, entre outras coisas, possuem funções e atributos necessários para a criação de caixas de diálogos, toolbars, tooltips e menus, baseados no sistema Windows. Esta biblioteca está incluída no ambiente de programação Visual C++. Ela possui dispositivos de ajuda (Wizards) para auxiliar a construção das janelas derivadas. A Autodesk, construiu bibliotecas baseadas no MFC que podem ser usadas para criar diálogos que operem e se comportem de forma parecida com o AutoCAD, porém, neste trabalho optou-se pela utilização das classes puras do MFC, por entender que estas fornecem um número maior de opções.

2.4 A Engenharia de Software

(Metodologia empregada pela Engenharia de Software no desenvolvimento de bons programas computacionais; contribuição deste trabalho no sentido de normatizar o desenvolvimento de programas dentro do Grupo CADTEC)

A engenharia de software é uma técnica criada para normatizar os parâmetros e os procedimentos a serem adotados ao se desenvolver um programa. Ela foi desenvolvida para auxiliar o programador a planejar e executar as diversas etapas constituintes da produção de um software. O desafio do engenheiro de software é escolher e montar as estruturas de grande complexidade que a programação permite realizar [Fil00].

Para desenvolver um software, segundo esta técnica, é necessário seguir o seguinte processo [Sta98]:

Etapa 1: Levantamento dos requisitos para o atendimento de uma determinada necessidade. Nesta etapa é de suma importância a completa integração entre o desenvolvedor e o cliente ou usuário [Fil00]. Uma boa especificação dos requisitos minimiza os custos de um projeto, pois aqui são definidos claramente os objetivos a serem alcançados.

Etapa 2: Estruturação das classes que conterão os objetos que melhor representarão a realidade do problema. Nesta etapa, cabe ao desenvolvedor extrair do mundo real, o modelo que possua características relevantes para o problema proposto. Durante os últimos anos, vários aplicativos foram desenvolvidos para auxiliar os programadores nesta tarefa. Tais programas utilizam a linguagem UML que pretende, através de diagramas, promover a visualização interativa das classes a fim de obter bons resultados, utilizando os paradigmas da programação orientada a objetos.

Etapa 3 : Estabelecer um cronograma de desenvolvimento planejado, com a definição seqüencial das diversas ações e, ao longo do desenvolvimento, fazer um acompanhamento regular através de um cronograma de desenvolvimento real.

Etapa 4: Confeccionar a documentação de todo o processo(Levantamento dos requisitos, análise, implementação, testes). Tais informações são necessárias para que uma pessoa diferente da que desenvolveu possa entender e alterar com sucesso o programa. Quando o trabalho é desenvolvido por um grupo, a engenharia de software

propõe a definição de normas de documentação claras e concisas, de forma a facilitar o intercâmbio de informações. Um código fonte pode ser documentado de várias formas. É interessante que, durante a digitação comentários relativos à seqüência de ações implementadas sejam introduzidos. Isto poderá auxiliar a descobrir de eventuais erros de lógica no programa. A definição de cabeçalhos de inicialização de arquivos, funções e variáveis ajuda a compreender o funcionamento do programa de forma global. Outra forma de produzir informações sobre o código fonte é lançar mão de aplicativos que constroem arquivos do tipo *help* no padrão Windows ou do tipo *HTML*, através da leitura de dados encontrados no código. Esta forma de documentação só pode ser gerada após a confecção do código.

Etapa 5 : Realizar testes durante todo o processo de desenvolvimento. Esta é uma facilidade encontrada na programação orientada a objetos, pois seus módulos podem ser compilados de forma independente e possuir comandos também independentes. Tais testes devem ser especificados e normatizados garantindo assim a sua qualidade e devem ser acompanhados de laudos relatando os *bugs* encontrados.

Etapa 6: Corrigir os erros encontrados. De posse dos laudos elaborados na Etapa 5 os desenvolvedores podem então planejar e executar as correções necessárias que também deverão ser rigorosamente documentadas e depois atualizadas.

Para implementar e usar os conceitos da engenharia de software no grupo CADTEC, foi necessário estabelecer algumas prioridades. Um dos primeiros objetivos foi integrar o grupo de desenvolvedores. Foi criado um site de integração e consulta no intuito de fornecer algumas normas para produção de uma boa documentação e estabelecer parâmetros gerais para nomear arquivos, classe, variáveis e funções. Tais normas e parâmetros foram estabelecidos conforme Arndt Von STAA [Sta98], e estão sofrendo alterações durante a sua utilização.(Ver site www.cadtec.dees.ufmg.br\engsoft). Esta iniciativa pretende ser um ponto de partida para a criação de uma biblioteca de classes do Grupo CADTEC.

Outra medida adotada foi a criação de um Grupo de Discussão como forma de promover a troca de idéias buscando a solução de problemas comuns ligados ao desenvolvimento dos aplicativos.

Colocar em prática os conceito da engenharia de software demanda muito trabalho e tempo. Porém, os resultados alcançados podem ser bastante satisfatórios. O

reaproveitamento de códigos proporcionado por esta prática permite um avanço na qualidade propiciando a construção de programas mais robustos e mais eficientes.

2.5 O CADTEC

(Descrição das atividades desenvolvidas no CADTEC (Centro de Apoio, Desenvolvimento Tecnológico e Ensino da Computação Gráfica) e sua contribuição para o desenvolvimento deste projeto)

O CADTEC (Centro de Apoio, Desenvolvimento Tecnológico e Ensino da Computação Gráfica) foi criado em 1997, pelo Departamento de Engenharia de Estruturas da UFMG, com o objetivo de desenvolver projetos multidiciplinares ligados à computação gráfica. Durante estes últimos anos, um grande número de pesquisadores, de diversas áreas de atuação, tiveram a oportunidade de fazer parte do grupo CADTEC.

A sua atuação tem se tornado bastante diversificada. Na área de desenvolvimento de aplicativos CAD, seu projeto piloto foi o desenvolvimento de um Modelador Estrutural [Mal98] e [Hüt98].Hoje se encontram em desenvolvimento três projetos nesta linha de pesquisa: o EletroCAD [Cot00], voltado para a concepção de projetos elétricos para edificações, o ArmaCAD, atendendo na área de detalhamento estrutural em concreto armado e o TowerCAD, que detalhará torres metálicas de transmissão. Outra contribuição recente foi o Modelador 3D para Vasos de Pressão [Bal00]. Existe também em andamento um projeto na área de geoprocessamento que utilizará ao aplicativo AutoCAD Map na automação do processo de edição de dados topográficos produzidos por estações GPS (Global Positioning System).

Vislumbrando as perspectivas abertas com a difusão da Internet os pesquisadores do CADTEC começaram a desenvolver sites para atender a diversos clientes dentro e fora da UFMG. Inicialmente desenvolvidos na linguagem html através de editores do tipo FrontPage, os sites ganharam nova funcionalidade com a utilização da ferramenta FLASH, que introduziu o som e o movimento.

Seguindo as novas tendências tecnológicas e de posse dos conhecimentos adquiridos na confecção de páginas eletrônicas, surgiu a vontade de produzir cursos

de educação à distância via Web. Tal iniciativa já produzir resultados, um Curso da Linguagem HTML , acessível em www.cadtec.dees.ufmg /cursohtml[Par00].

Visando a capacitação de seus integrantes, o grupo CADTEC procura promover e participar de vários eventos para discussão e aprendizado nos diversos ramos da computação gráfica. Seminários, palestras e cursos abordando assuntos como Programação em ObjectARX, Engenharia de Software e Design para Web, são alguns exemplos das atividades desenvolvidas.

Dentro do seu objetivo de se tornar um centro de difusão de conhecimentos, o CADTEC promove cursos para o público interno e externo à Universidade. Os Alunos de Linguagem C++, ObjectARX, AutoCAD Básico e Avançado, contam com professores bem preparados ligados ao CADTEC e recebem materiais didáticos que também é produzido pelo grupo.

Visando a integração entre Empresa e Universidade, o grupo CADTEC tem realizado parcerias com empresas privadas no intuito de conseguir suporte para o desenvolvimento de novas tecnologias. A primeira empresa a participar foi a Autodesk. Ao transformar o grupo CADTEC em ADN(Autodesk Development Network) foram abertas possibilidades de acesso a diversos serviços, tais como versões atualizadas do pacote AutoCAD e uma ligação online para esclarecimento de dúvidas em site exclusivo. Um aparato completo que ajuda desenvolver e operar com segurança o AutoCAD.

Empresas interessadas no desenvolvimento de tecnologia CAD, também querem fazer parceria com o CADTEC. Esta foi uma forma encontrada para aliar o trabalho acadêmico com as necessidades da empresa. Confiando neste trabalho a emprese mineira CODEME Estruturas Metálicas Ltda. participou do projeto "Modelador Estrutural" [Mal98] e [Hüt98] e a empresa mineira CR Gontijo Ltda. está participando do projeto "TowerCAD".

São inúmeras as oportunidades oferecidas, porém, infrutíferos seriam os resultados se não houvesse o espírito de cooperação e responsabilidade, cultivado por cada um que passou por ali. Esta pode ser a chave do sucesso.

Capítulo 3

PLANEJAMENTO DO DESENVOLVIMENTO DO APLICATIVO

Escopo Conceitual

Fluxograma de Desenvolvimento

3.1 Escopo Conceitual

(Levantamento dos requisitos necessários para detalhar uma estrutura em concreto armado via CAD; definição os objetos que compõem o universo estudado através da abstração correta dos seus componente)

3.1.1 Levantamento dos Requisitos:

Inicialmente este aplicativo seria desenvolvido para estruturas pré-moldadas de concreto. Tais peças, por serem parametrizadas, tornam-se muito atrativas a este tipo de desenvolvimento. Porém, não foi possível realizar parcerias com as empresas do ramo. No intuito de aproveitar a experiência profissional anterior, optou-se então pela elaboração de um aplicativo para atender ao detalhamento de peças de concreto armado da estrutura convencional.

Ao se pensar em uma estrutura convencional em concreto armado, podemos extrair alguns componentes básicos: estruturas de fundação (blocos, tubulões e estacas), lajes, vigas e pilares. Todos os componentes deverão ser representados em conjunto e separadamente através dos seus desenhos de fôrma e armação. Baseado na experiência chegou-se a conclusão que as peças mais trabalhosas para se detalhar eram as vigas e os pilares, devido à quantidade maior de unidades encontradas na estrutura e a infinidade de variantes encontradas na representação de tais modelos.

O processo de detalhamento de uma peça em concreto armado obedece às seguintes etapas:

1. Representação da caixa da peça: Esta é a forma geométrica que a peça irá tomar depois de ser concretada. As dimensões para a confecção desta caixa serão determinadas na etapa de dimensionamento e inseridas no desenho de fôrma do pavimento em que ela se encontra. O desenho da fôrma já deve ter sido confeccionado anteriormente.
2. Representação da armadura da peça: Através da análise dos esforços solicitantes, a ferragem é definida, combatendo as tensões de tração e compressão. O desenho das armaduras que irão compor a ferragem é o verdadeiro propósito do detalhamento das armaduras em concreto armado. É ali que o engenheiro responsável pela execução da obra irá encontrar

todas as informações para fazer o corte, a dobra e a colocação da armação dentro da caixa de madeira que será concretada

3. Confecção da lista de materiais: É usual quando se executa um projeto contar com listas quantitativas das peças componentes. Isto facilita a elaboração dos orçamentos financeiros e a posterior compra do material necessário. No caso do projeto estrutural, é necessário quantificar a ferragem. Uma peça de concreto contém em seu interior várias barras de ferro em posições diferentes. Cada uma delas recebe um número que a identifica. Depois de numerar todas as barras, elas devem ser listadas segundo a sua bitola, o seu comprimento e a quantidade de vezes em que ela aparece no desenho. Isto deve ser feito para cada peça em separado e depois, todos os comprimentos resultantes devem ser somados formando uma tabela que fornecerá a quantidade em quilos da ferragem necessária.

As estruturas de concreto armado podem assumir inúmeras formas geométricas. Porém, ao automatizar um processo, deve-se começar por definir quais são os modelos mais encontrados na prática. Em estruturas convencionais, as vigas e os pilares geralmente têm seção transversal retangular. Na maioria das vezes esta seção e o seu eixo permanecem constantes ao longo de toda a peça.

Uma peça em 2D pode ser representada de várias formas. Ela pode ser desenhada em vista, em corte ou em planta. Usualmente, ao se detalhar uma viga em concreto armado pode-se desenhá-la em corte, quando sua armadura é constituída por uma ferragem reta ou pouco complexa. Este tipo de representação é mais utilizado para viga constituídas de apenas um tramo [**Figura 3-1**].

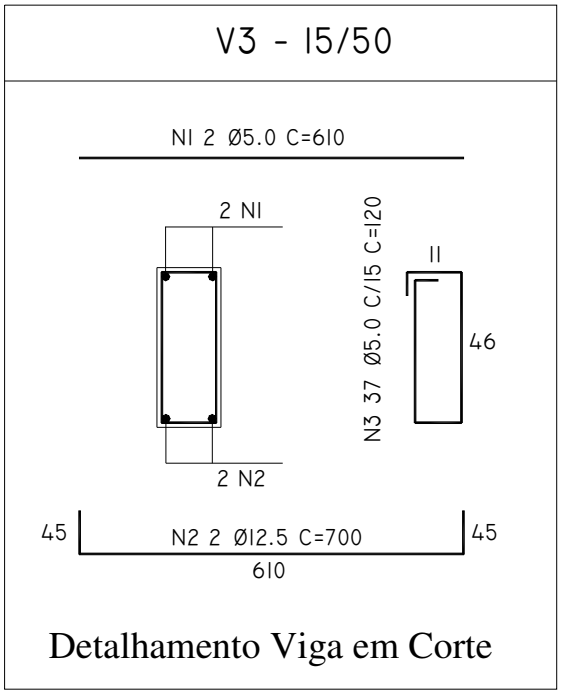


Figura 3-1

Outra forma de representação, talvez a mais utilizada, é a vista. Ela permite a melhor visualização das vigas que possuem vários tramos e é capaz de demonstrar com mais clareza o detalhamento de ferragens mais complexas. Ela também permite visualizar a posição correta da ferragem e das dobras nela existentes [**Figura 3-2**].

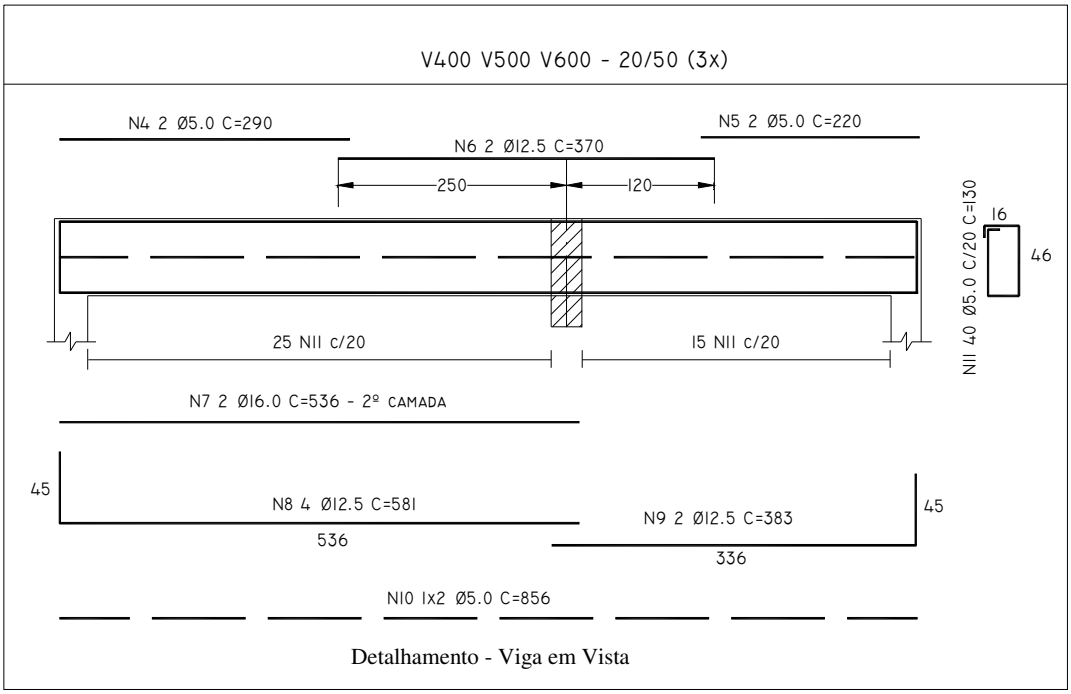


Figura 3-2

Para o caso dos pilares, a representação é em corte transversal. É muito comum ocorrer mudanças de dimensão de um pilar quando este passa de um andar para outro. Isto ocorre porque, quanto mais alto for o andar, menor é o peso sustentado pelo pilar, que pode ter então a sua seção transversal reduzida. Porém, tais pilares geralmente permanecem com a mesma seção se pensarmos em cada elevação em separado.[**Figura 3-3**].

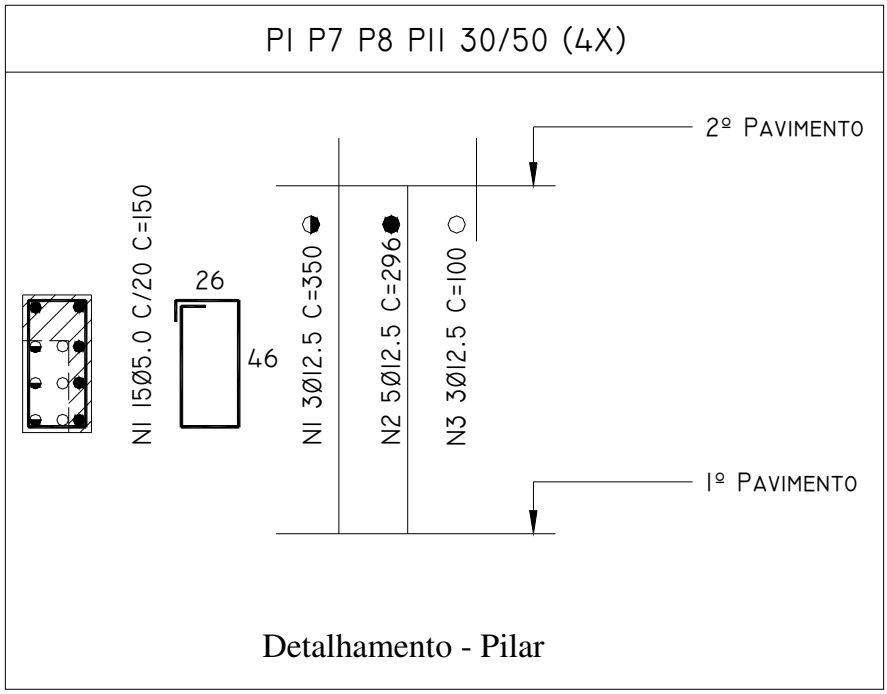


Figura 3-3

Depois de definir o tipo e a forma da peça a ser detalhada pelo aplicativo, deve-se pensar um pouco sobre o desenho de sua armadura. Muitos consideram que detalhar uma peça é uma arte. Dependendo do engenheiro de estruturas, uma mesma viga pode ser armada de maneiras diferentes. Isto acontece porque existem várias formas de combater os esforços a que estão submetidas as peças de concreto armado. Mas existe um consenso; cada um normatiza seus procedimentos de modo a manter uma certa coerência ao longo do projeto.

Em uma viga, podemos destacar vários tipos de armaduras: armadura superior, armadura inferior, estribos e costelas. Cada uma delas possui uma representação gráfica e encontra-se posicionada em um local diferente dentro da caixa. Tais armaduras podem ainda ser dobradas ou retas, com ou sem gancho e ainda possuir ou não um comprimento pré-definido. Deve-se lembrar ainda que elas geralmente são vendidas em barras com 11 metros de comprimento. Portanto, em vigas e pilares que

necessitem de comprimentos maiores, tais barras devem sofrer emendas. Todos estes tipos de armação se desdobram em outros com representações distintas. Por exemplo: uma armadura inferior de uma viga pode ser formada por barras sem dobra ou dobradas a 45º, 60º, ou 90º graus. [**Figura 3-4**].

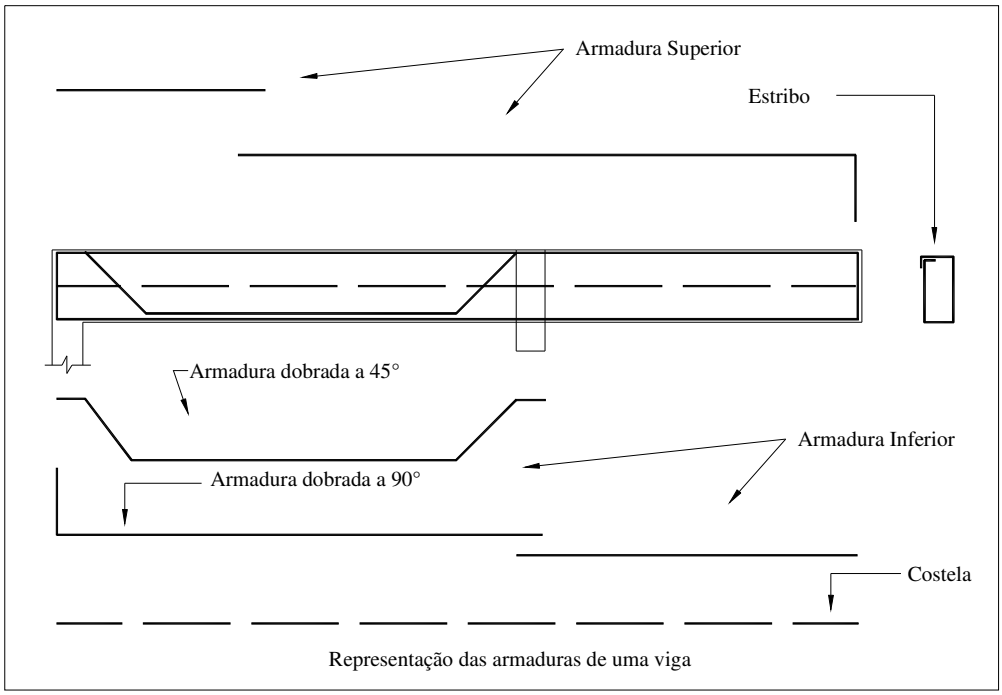


Figura 3-4

Já para os pilares, os tipos de armadura são: armação que morre, armação que continua, esperas, estribos, e espaçadores. Cada uma delas também apresenta uma representação distinta e uma posição diferente na caixa. **[Figura 3-5]**

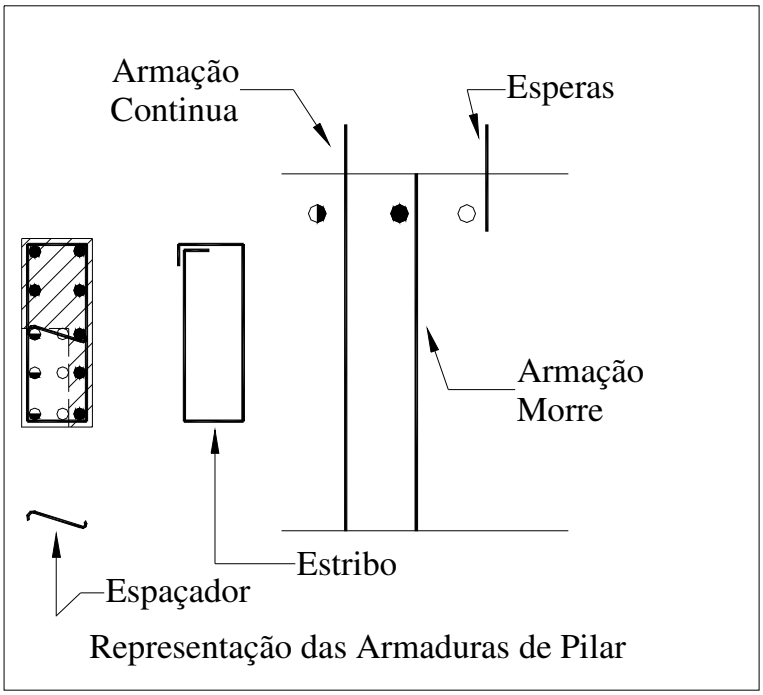


Figura 3-5

Conforme a área de ferro necessária para combater os esforços, as dimensões da peça e as necessidades construtivas o engenheiro poderá escolher qual a bitola mais adequada a ser usada. Cada uma terá seu comprimento de ancoragem estabelecido pela NBR 6118, que é calculada de acordo com as tensões máximas adotadas para o concreto e o aço [Sus85].

A confecção da lista de materiais se dará através da coleta dos dados definidos para cada barra depois que todas estiverem devidamente numeradas. Caso alguma barra seja apagada ou modificada, a lista deve acompanhar esta troca.**[Figura 3-6]**

LISTA DE FERROS				
POS.	QUANT.	Ø	COMPRIMENTO UNITÁRIO (cm)	COMPRIMENTO TOTAL (cm)
N1	2	10.0	320	640
N2	6	5.0	290	1740
N3	5	16.0	720	3600
N4	25	5.0	150	3750
N5	2	12.5	380	760
N6	5	12.5	520	2600
N7	7	6.3	90	630
N8	2	10.0	CORR.	2530
N9	45	5.0	170	7650
N10	3	16.0	320	960

RESUMO			
TIPO DE AÇO	Ø	COMPRIMENTO TOTAL (m)	PESO (kg)
CA 50	5.0	14	34
	6.3	7	2
	10.0	32	21
	12.5	34	34
	16.0	46	74
TOTAL GERAL (kg)			165

Lista de Materiais

Figura 3-6

Depois de traçar o panorama do problema, pode-se definir os requisitos necessários para produzir um desenho de detalhamento de estrutura de concreto armado via CAD.

- O usuário deverá ser capaz de construir e editar representações em 2D da vista e/ ou corte da caixa de vigas e pilares com seção transversal retangular e com o eixo geométrico constante. Elas serão desenhadas inicialmente na escala 1:1 do AutoCAD.
- O Programa deverá fornecer mecanismos para que o usuário escolha o tipo de barra, sua bitola e sua posição, de forma a combater os esforços calculados.
- O Programa deverá ser capaz de calcular o comprimento das barras para cobrir o diagrama de momento fletor, o seu comprimento de ancoragem e o seu raio de curvatura de forma a poder representá-las corretamente.
- O Programa deverá gerar a lista de materiais, a todo o momento que o usuário solicitar, de forma automática.

3.1.2 Estruturação das classes:

Abstraindo os objetos que compõem o universo estudado, pode-se perceber a existência de três módulos: o Módulo 1 que representará a Caixa da Peça, o Módulo 2 representado pelas Barras de Ferro e o Módulo 3 que conterá a Lista de Materiais. É fácil perceber que todos estão interligados e por diversas vezes, um módulo precisa de informações (dados) que estão contidos em outro módulo.

Ao se fazer uma abstração de dados, agora voltada para cada universo dos grupos citados, podemos obter uma série de objetos derivados que representarão as entidades já comentadas.

Aplicando um dos princípios fundamentais da Orientação a Objetos, foi definida a hierarquia de classes a ser seguida durante o desenvolvimento deste projeto **[Figura 3-7]**.

A classe CCaixa será tratada como uma classe "Pai". Este tipo de classe conterá os atributos comuns que serão utilizados pelas classes filhas na sua implementação. Todos os seus membros foram declarados *public* (públicos).

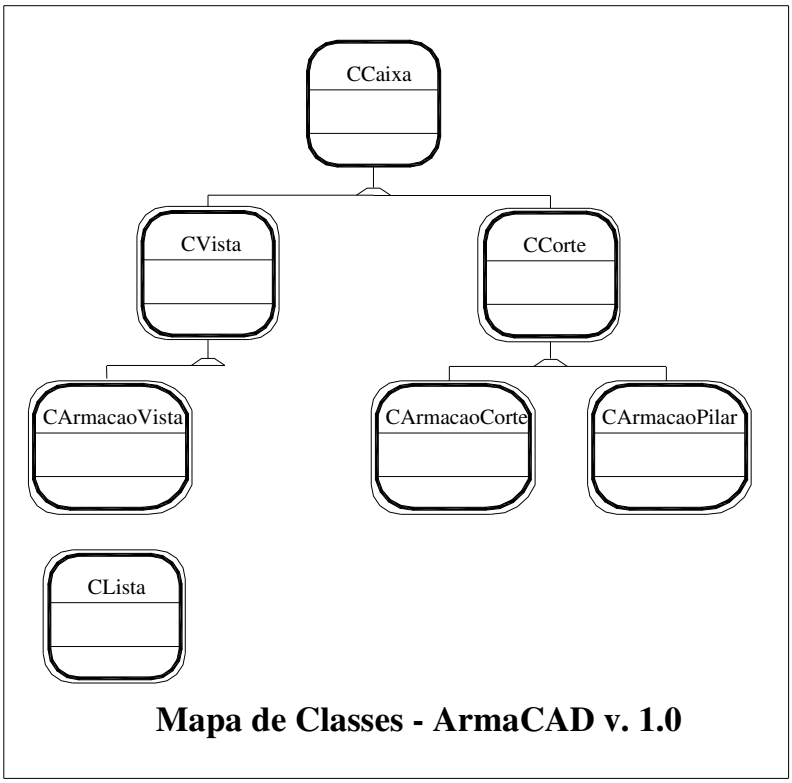


Figura 3-7

Logo abaixo, aparecem as classes CVista e CCorte. Elas serão responsáveis pela representação gráfica em 2D da caixa da peça em vista e em corte. CVista irá conter os objetos do tipo viga que serão representados em VISTA e CCorte deverá conter os objetos do tipo pilar e do tipo viga representados em CORTE. Depois de construir a fôrma, deve-se então lançar mão das classes CArmaçãoVista, CArmaçãoCorte e CArmaçãoPilar, que implementarão os diversos tipos de armadura. Finalmente, os dados já fornecidos poderão ser utilizados na montagem da lista de materiais através da classe CLista. Todos os membros de todas as classes foram declarados públicos (*public*) de modo a serem utilizados para construir os diversos objetos presentes na hierarquia de classes.

A classe CCaixa herdará as características da classe AcDbEntity. Esta classe pertence à biblioteca ObjetcARX e que é responsável pelo armazenamento dos dados que são necessários para representar graficamente o objeto.

Os atributos e métodos de cada classe [**Figura 3-7**], foram sendo definidos e implementados de acordo com um fluxograma de desenvolvimento.

3.2 Fluxograma de Desenvolvimento

(Programação das etapas de desenvolvimento em módulos)

Com o programa concebido em módulos, fica fácil perceber como ele será o seu fluxograma de desenvolvimento. A **Figura 3-8** mostra de forma sistemática como serão as etapas de desenvolvimento.


Análise da Classe CCaixa
Implementação da Classe "Pai" CCaixa, Testes
Documentação
Análise da Classe CCorte
Implementação da Classe&Objeto CCorte, Testes
Documentação
Análise da Classe CVista
Implementação da Classe & Objeto CVista, Testes
Documentação
Análise Interface CVista e CCorte
Implementação das interfaces para as classes CCorte e CVista, Testes
Documentação
Análise de CArmacaoCorte
Implementação da Classe&Objeto CArmacaoCorte, Testes
Documentação
Análise de CArmacaoVista
Implementação da Classe&Objeto CArmacaoVista, Testes
Documentação
Análise de CArmacaoPilar
Implementação da Classe&Objeto CArmacaoPilar, Testes
Documentação
Análise Interface CArmacaoCorte CArmacaoVista e CArmacaoPilar
Implementação da interface das classes CArmacaoCorte, CArmacaoVista, CArmacaoPilar, Testes
Documentação
Análise da Classe CLista
Implementação da Classe &Objeto CLista, Testes
Documentação
Análise Interface de CLista
Implementação da interface da classe CLista, Testes
Documentação
Fluxograma de Desenvolvimento

Figura 3-8

Capítulo 4

DESCRIÇÃO DOS MÓDULOS DO APLICATIVO ARMACAD

Aspectos comuns ao declarar e implementar uma classe costumizada

O Módulo Caixa

O Módulo Armação

O Módulo Lista

4.1. Aspectos comuns ao declarar e implementar uma classe costumizada

(Descrição das macros e funções indispensáveis para o perfeito funcionamento das classes derivadas da Biblioteca ObjetcARX)

Uma classe costumizada é uma entidade personalizada do AutoCAD. Através de um aplicativo *.arx* pode-se controlar o seu comportamento gráfico, lógico e funcional.

O ObjectARX possui um conjunto de macros que ajudam a criar estas classes derivadas. São elas:

- **ACRX_DXF_DEFINE_MEMBERS(CLASSE_NAME, PARENT_NAME, DWG_VERSION, MAINTENANCE_VERSION, PROXY_FLAGS, DXF_NAME, APP)** : Esta macro tem como função registrar a classe costumizada na árvore hierárquica do AutoCAD , definindo seu objeto descritor (*class descriptor object*), para que ele possa ser identificado em tempo de execução. Ela deverá ser inserida no início do arquivo *.cpp* da implementação da nova classe.

Exemplo:

```
#include "stdafx.h"

ACRX_DXF_DEFINE_MEMBERS(CCaixa, AcDbEntity,
      AcDb::kDHL_CURRENT, AcDb::kMReleaseCurrent, 0,
      CCaixa, /*MSG0*/ "AutoCAD");
```

- **ACRX_DECLARE_MEMBERS(CLASSE_NAME)** : É uma macro do ObjectARX que declara três funções que ajudam no reconhecimento das classes derivadas. A função *desc()* que retorna o objeto descritor da nova classe, a função *cast()* que retorna um objeto de determinado tipo ou NULL se este objeto não existir e a função *isA()* que retorna a classe correspondente a um objeto desconhecido. Esta Macro deverá ser inserida na seção pública da declaração da nova classe, no arquivo *.h*.

Exemplo:

```
class CCaixa : public AcDbEntity
{
      public:
            ACRX_DECLARE_MEMBERS(CCaixa);
            ..........
}
```

- **MAKE_ACDBOPENOBJECT_FUNCTION (CLASS_NAME)** : Esta macro contém funções que irão abrir o banco de dados do AutoCAD. Ela deve ser inserida no final do arquivo *.h* da nova classe.

Exemplo:

```
MAKE_ACDBOPENOBJECT_FUNCTION(CCaixa);
#endif
```

Uma classe e seus comandos deverão ser inicializados para serem usados pelo aplicativo dentro do AutoCAD. Na função *initAPP()*, a inicialização das classes se dá através da função *rxInit()* e a inicialização dos comandos é feita pela função *addComand()*, da classe *acedRegCmds*. O primeiro parâmetro desta função, pede o nome do grupo a que o comando pertence. Isto facilita da sua posterior remoção, quando o aplicativo for descarregado. No exemplo abaixo, o comando adicionado pertence ao grupo "CAIXA_COMMANDS".

Exemplo:

```
void initAPP()
{
      acedRegCmds->addCommand("CAIXA_COMMANDS",
      "ARMACAD_PARAMETROS", "PARAMETROS", ACRX_CMD_MODAL,
      SetaParametrosDimensionamento);

      CCaixa::rxInit();
      CCorte::rxInit();
      CVista::rxInit();
      CArmacaoCorte::rxInit();
      CArmacaoVista::rxInit();
      acrxBuildClassHierarchy();

}
```

As classes deverão ser mencionadas na ordem hierárquica em que aparecem no mapa de classes do aplicativo. A função *acrxBuildClassHierarchy()* será chamada ao final da inicialização das classes derivadas de forma a inseri-las no mapa de classes do AutoCAD.

Quando a aplicação for encerrada, através da função *unloadApp()* , todos os comandos e classes derivadas inicializados deverão ser removidos. Para retirar as classes derivadas usa-se a função *deleteAcRxClass()*. Para remover grupos de comando inseridos pelo aplicativo usa-se a função *removeGroup()* pertencente a classe *acedRegCmds*.

Exemplo:

```
void unloadApp()
{

      acedRegCmds->removeGroup("CAIXA_COMMANDS");
      deleteAcRxClass(CArmacaoVista::desc());
      deleteAcRxClass(CArmacaoCorte::desc());
      deleteAcRxClass(CVista::desc());
      deleteAcRxClass(CCorte::desc());
      deleteAcRxClass(CCaixa::desc());

}
```

É importante observar que as classes deverão ser removidas na ordem inversa da qual foram inseridas.

Em uma classe costumizada são implementadas novas funções membro e/ou elas são sobrecarregadas da classe pai. Estas funções irão manipular dados, podendo modificar o seu valor ou apenas utilizá-los em suas rotinas. Usualmente, as funções *assertReadEnable(), assertWriteEnabled(), assertNotifyEnable(),* são chamadas no início das funções membro da nova classe para que se possa testar o estado do objeto (leitura, escrita, notificação). Se por exemplo a função for apenas utilizar os valor dos atributos armazenado no banco de dados, deve-se assegurar que tais valores não sejam modificados, com a chamada a função *assertReadEnable().* Se o objeto for aberto sem a definição do seu estado, o AutoCAD é automaticamente fechado.

Através dos comandos registrados no AutoCAD , o usuário poderá fornecer os valores para os diversos atributos que o objeto possui. Cada um deles deverá ser armazenado dentro do Banco de Dados. Para que se possa acessar, modificar e gravar estes dados, foram desenvolvidas duas funções : *m_SetXXX()*, que guarda no Banco de Dados os valores fornecidos pelo usuário e *m_GettXXX(),* capaz de recuperar o valor armazenado, quando solicitado pelo aplicativo. Elas serão funções membro públicas (*public*) de suas respectivas classes, podendo ser acessadas por suas classes derivadas.

Exemplo:

```
/* Função membro da classe CCaixa capaz de recuperar o
valor da Largura armazenado no Banco de dados*/
double CCaixa::m_GetLarg(void)
{
      assertReadEnabled();
      return m_dLarg;
}


/*Função membro da classe CCaixa capaz de registrar o
valor da Largura no Banco de Dados.*/
void CCaixa::m_SetLarg(const double val1)
{
      assertWriteEnabled();
      m_dLarg=val1;
      recordGraphicsModified();
}
```

É óbvio que todos os dados, que precisam ser armazenados para posterior resgate ou modificação, deverão possuir a sua função *m_SetXXX()* e *m_GetYYY()*.

Como já foi dito, a classe AcDbObject possui funções capazes de armazenar os dados quando se salva ou abre um desenho do AutoCAD. Elas são funções virtuais, isto é, possuem implementações diferentes para cada classe derivada de AcDbObject. O seu principal argumento é um ponteiro para a classe AcDbDwgFiler. O ponteiro **filer* para o objeto desta classe representa um "espaço de memória" no banco de dados onde será armazenado o valor do atributo.

O exemplo abaixo demonstra como deve ser a implementação das funções *dwgInFiels()* e *dwgOutFiels()*. Primeiramente, deve-se assegurar de que o objeto seja aberto no estado correto. Para tal usa-se as funções *assertWriteEnabled()* e *assertReadEnabled()*. O próximo passo é chamar a própria função para a classe pai, caso contrário resultará em erro de execução. O importante é observar que a ordem em que os atributos estão listados para leitura e escrita deve ser sempre a mesma. Se o programa possuir um dado do tipo inteiro, deve-se especificar o seu tamanho, por exemplo, *readInt16*, *writeInt32*.

```
Virtual Acad::ErrorStatus
dwgInFields(AcDbDwgFiler* filer).
{
      assertWriteEnabled();//objeto aberto para escrita
      AcDbEntity::dwgInFields(filer);// chamada da função - classe
      //pai
      filer->readItem(&m_dLarg);//leitura do valor do Atributo
      filer->readInt16(&m_iTipoCorte);// Especificação do tamanho do
      //inteiro
      return filer->filerStatus();
}
```

```
Virtual Acad::ErrorStatus
dwgOutFields(AcDbDwgFiler* filer).
{
      assertReadEnabled();//objeto aberto para leitura
      AcDbEntity::dwgOutFields(filer);// chamada da função – classe
      //pai
      filer->writeItem(&m_dLarg);//gravação do valor do atributo
      filer->writeInt16(&m_iTipoCorte);// Especificação do tamanho do
      //inteiro
      return filer->filerStatus();
}
```

4.2 O Módulo Caixa

(Descrição do primeiro módulo, destacando os membros de cada classe derivada)

4.2.1 A classe CCaixa

A classe CCaixa, embora não possua representação gráfica, herdará as características de AcDbEntity. Nela deverão estar encapsulados todos os métodos e atributos comuns encontrados nas classes filhas CVista e CCorte.

Para definir os atributos que irão compor esta classe é indispensável abstrair os dados do universo estudado. Como já definido, a seção transversal das peças a serem representadas terá forma retangular e constante. A geometria da peça exige informações como a **Largura** e a **Altura**.

Uma peça do tipo Viga pode ser representada em Vista e/ou em Corte. No caso da representação em vista, podemos ter uma viga com vários tramos. A idéia é tratar cada tramo em separado, destacando seu vários componentes: **Largura do Apoio a Esquerda, Vão, Largura do Apoio a Direita**. No caso da representação de uma Viga em Corte, estes valores não contribuem para a representação gráfica do modelo mas serão importantes na hora de definir o tamanho das barras da armadura. Um outro dado seria a quantidade de vãos que a viga possui, mas este é exclusivo da representação em vista [**Figura 4-1**].

Um Pilar será sempre representado em Corte. Neste caso, além da largura e altura da seção transversal, devemos fornecer o tamanho do pé direito, dado exclusivo da classe CCorte e será introduzido lá.

Cada peça deverá ser identificada com um **Nome** e nº de **Unidades** existente da mesma peça, de acordo com a planta de fôrma.

Quando se quer criar uma linha no AutoCAD a primeira medida a ser tomada é fornecer o primeiro ponto. Isto ocorre com todos os comandos de desenho deste programa. Uma entidade costumizada também necessita das coordenadas do seu primeiro ponto ou **Ponto de Inserção**. A partir dele e dos dados necessários para construir o desenho, todos os outros pontos serão encontrados. O **Ponto de Inserção** poderá ser obtido clicando na tela ou fornecendo suas coordenadas.

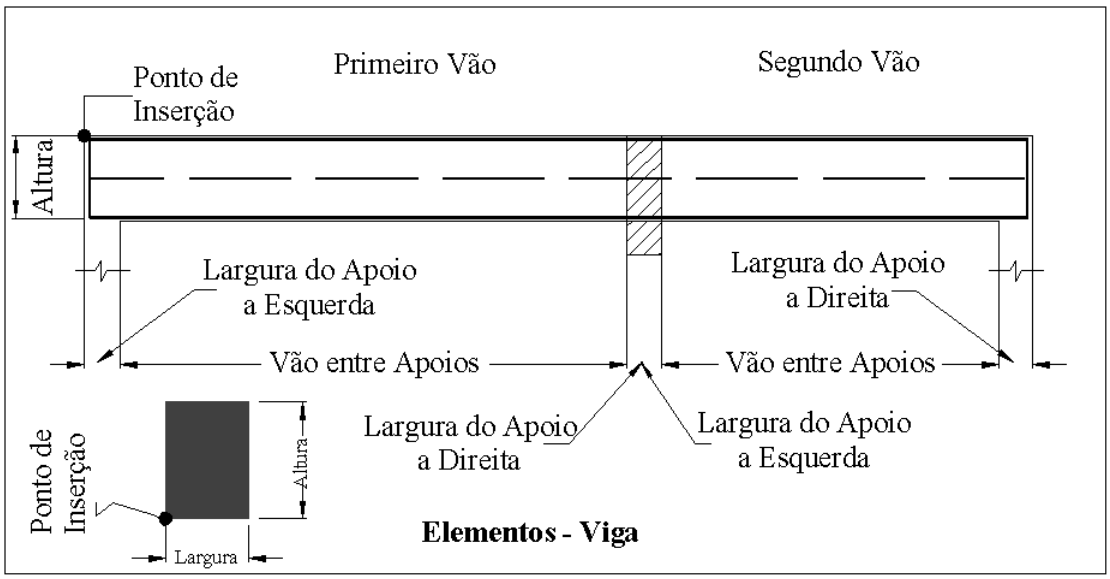


Figura 4-1

O ensaio realizado em barra de aço mergulhada num bloco de concreto, tracionada por uma força que tende a arrancá-la da massa de concreto, demonstrou que a barra de aço desliza sobre a massa de concreto. Tal força deverá ser combatida pela tensão de aderência que se manifesta entre o perímetro da barra e o concreto que a envolve.[**Figura 4-2**] Alguns parâmetros de projeto deverão ser fornecidos para determinar do comprimento de ancoragem das barras, de forma a não permitir o seu deslizamento dentro da massa de concreto. São eles: **Resistência Característica do Concreto à Compressão(f_{ck}) Resistência de Escoamento do Aço à Tração(f_y), Coeficiente de Minoração da Resistência do Concreto e Coeficiente de Minoração da Resistência do Aço.** Tais parâmetros farão parte de escopo da Classe CCaixa pois serão os primeiros dados fornecidos, assim que o aplicativo for inicializado. Posteriormente serão utilizados pelo módulo ARMAÇÃO para definir o comprimento de ancoragem das barras de aço componentes.

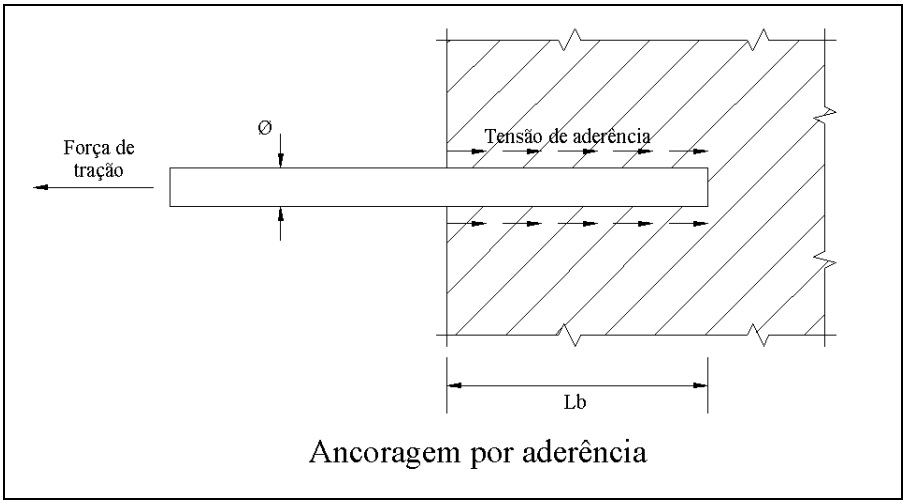


Figura 4-2

Agora, pode-se enumerar os atributos componentes da classe CCaixa :

Nome do atributo	**Descrição do que representa**
m_dLarg	Largura da seção transversal retangular
m_dAlt	Altura da seção transversal retangular
m_dApoioE	Largura do apoio da viga à esquerda
m_dVão	Comprimento do vão da viga entre apoios
m_dApoioD	Largura do apoio da viga à direita
m_scNomePeça	Nome da peça a ser detalhada
m_iUnidades	Quantidade de peças iguais e com a mesma armação, encontradas no desenho de fôrma
m_cordPontoInsertCaixa	Ponto de Inserção da entidade na tela do AutoCAD
m_dTensãoAco	Resistência Característica do Aço à tração
m_dTensãoConcreto	Resistência Característica do Concreto à compressão
m_dCoefMinoraAco	Coeficiente de minoração da resistência do aço à tração
m_dCoefMinoraConcreto	Coeficiente de minoração da resistência do concreto à compressão

Tabela 4-1 Atributos da classe CCaixa

Os Métodos Próprios da classe CCaixa serão aqueles capazes de obter e modificar os valores dos atributos, e já foram mencionados no item 5.1. Outras funções pertencentes ao escopo da classe AcDbEntity e de sua classe pai AcDbObject, também deverão ser sobrecarregadas e serão utilizadas por suas classes filhas. A princípio, serão sobrecarregadas as seguintes funções virtuais:

Nome da Função Sobrecarregada	**Descrição**
virtual Acad::ErrorStatus dwgInFields(AcDbDwgFiler* filer)	Guarda o valor dos atributos no banco de dados do arquivo *.dwg* quando se salva um desenho no AutoCAD.
virtual Acad::ErrorStatus dwgOutFields(AcDbDwgFiler* filer)	Resgata o valor dos atributos guardados no arquivo .dwg quando se abre um desenho no AutoCAD.
virtual Adesk::Boolean wordDraw(AcGiWorlDraw* mode)	Responsável por construir a representação gráfica da entidade.
virtual Acad::ErrorStatus getGripPoints(AcGePoint3dArray& gripPoints, AcDbIntArray& osnapModes, AcDbIntArray& geomIds)const	Responsável pela definição da localização dos pontos de grip (grip points) definidos para a entidade costumizada
virtual Acad::ErrorStatus moveGripPointsAt(const AcDbIntArray& indices, const AcGeVector3d& offset)	Responsável pelos comandos de grip “stretch" e "move” do AutoCAD ""
virtual Acad::ErrorStatus transformBy(const AcGeMatrix3d& xform)	Responsável pela aplicação das matrizes de transformação para copiar, movimentar e escalonar a entidade

Tabela 4-2 - Métodos Sobrecarregados da classe CCaixa

4.2.2 A classe CCorte

A Classe&Objeto CCorte [**Figura 3-7**], derivada da classe CCaixa, é responsável pela representação gráfica de Vigas e Pilares em Corte.

Considerando a seção transversal das peças a serem desenhadas que, em ambos os casos, será retangular, os elementos **Largura** e **Altura** deverão estar presentes. Tais atributos serão herdados da classe CCaixa. Porém, apesar de possuírem a mesma seção em Corte, vigas e pilares precisarão armazenar dados diferentes extraídos do desenho de fôrma.

Para as Vigas, os atributos necessários **Largura do Apoio Esquerdo** , **Vão**, **Largura do Apoio Direito**, já foram declarados na classe CCaixa, e dela serão herdados.

O detalhamento das armaduras para o elemento estrutural Pilar é executado tomando-se cada pavimento em separado, portanto a **Altura do Pé Direito** é um dado necessário. O pilar pode sofrer transição de um andar para o outro. Esta transição poderá ocorrer de várias formas. A parcela da seção do pilar que será reduzida no pavimento superior deverá aparecer hachurada [**Figura 4-3**].

Além da **Largura** e **Altura** da peça, outros atributos serão requeridos para construir esta representação gráfica: **Largura do Pilar no Pavimento Superior**, **Altura Pilar no Pavimento Superior**. De posse destas novas dimensões, basta agora fornecer sua nova posição em relação ao Ponto de Inserção inicial. Ao fornecer o **Deslocamento Horizontal** e **Deslocamento Vertical** do referido ponto teremos a nova posição do Pilar no andar superior. [**Figura 4-4**].

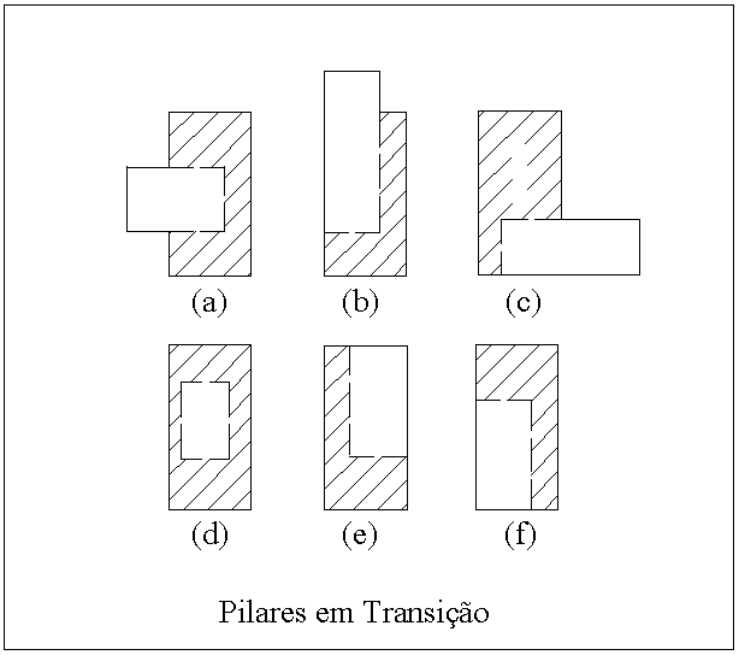


Figura 4-3

Observando melhor a **Figura 4-3** pode-se notar que as transições que ocorrem em (a),(b) e (c) , são menos comuns do que as apresentadas em (d), (e), e (f). No primeiro caso, o pilar superior se desloca para fora da região ocupada pelo pilar inferior. Para evitar problemas estruturais, tal parcela deverá "nascer" em cima de um outro elemento estrutural como uma viga, por exemplo.Ao modelar uma estrutura, o engenheiro procura evitar este tipo de transição, pois a adoção deste tipo de solução pode comprometer a estabilidade da estrutura..

Sempre privilegiando situações que na prática se apresentem comuns, optou-se por permitir a transição de um pilar, somente se o deslocamento em qualquer uma das direções (x e y), não ultrapassar os limites estabelecidos no pavimento inferior.

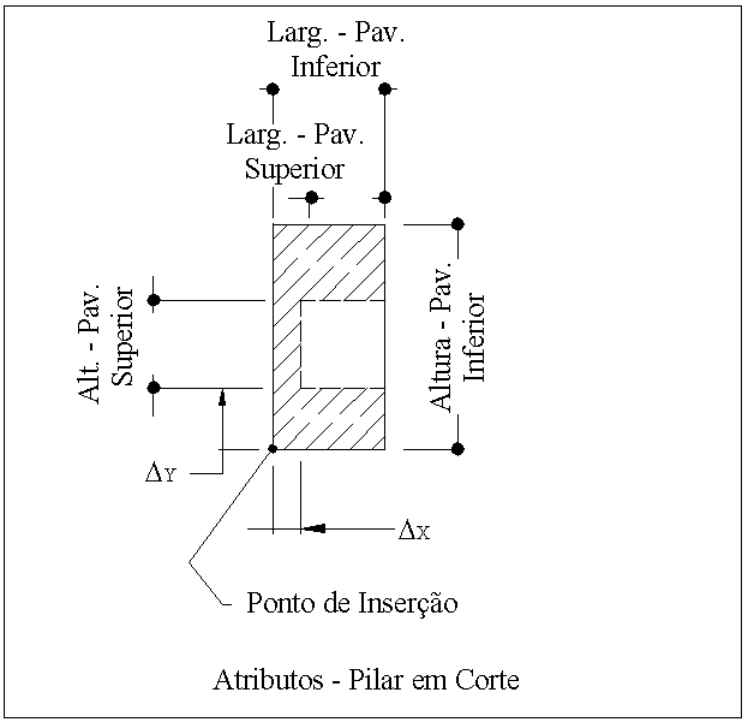


Figura 4-4

A classe CCorte possui portanto dois tipos de objetos, com representações gráficas distintas e armazenando informações diferentes. Para que o aplicativo possa distingui-los foi criado o atributo do tipo inteiro *m_iTipoCorte* que assumirá o **valor 1**, se o tipo da peça for uma Viga e o **valor 2** se o tipo da peça for um Pilar.

Uma hachura, é uma entidade nativa do AutoCAD, e pertence à classe **AcDbHatch**. Para que ela possa aparecer no desenho, faz-se necessário declarar um ponteiro **pHatch*, de forma a utilizar o objeto do tipo AcDbHatch para criar objetos do tipo CCorte.

As variáveis atribuídas a esta classe serão, portanto:

Nome do atributo	**Descrição do que representa**
m_dPeDireito	Altura medida entre o pavimento inferior e o pavimento superior
m_dLargPavSup	Largura do pilar no pavimento superior
m_dAltPavSup	Altura do pilar no pavimento superior
m_dDeltaX	Deslocamento na direção X, sofrido pelo pilar no pavimento superior.
m_dDeltaY	Deslocamento na direção Y, sofrido pelo pilar no pavimento superior.
m_scNomePeça	Nome da peça a ser detalhada
m_itipoTransicao	Variável que identifica qual tipo de transição de pilar desenhada.
*pHatch	Ponteiro para um objeto da classe AcDbHatch, responsável pela deteminação dos parâmetros necessários para desenhar hachuras no AutoCAD.

Tabela 4-3 - Atributos da classe CCorte

A Classe&Objeto CCorte, além de possuir as funções do tipo *m_GetXXX* e *m_SetYYY* para cada um de seus atributos, terá métodos próprios que ajudarão a construir a representação gráfica dos seus objetos. São elas:

Nome da Função Membro	**Descrição**
virtual Acad::ErrorStatus m_GetVertices (AcGePoint3dArray& cordVertices)const	Constrói um "array" contendo as coordenadas dos vértices da seção retangular, partindo do Ponto de Inserção definido pelo usuário.
Adesk::Boolean m_DesenhaCorteViga (AcGiWorldDraw* wd)	Constrói, através do ponteiro **wd*, o objeto que representa a seção transversal da Viga. (Desenho da seção transversal, texto, linhas auxiliares, habilitação de layers).
Adesk::Boolean m_DesenhaCortePilar (AcGiWorldDraw* wd);	Constrói, através do ponteiro **wd*, o objeto que representa a seção transversal do Pilar. (Desenho da seção transversal do pavimento inferior e do pavimento superior, hachuras, linhas auxiliares, textos, habilitação de layers).
Adesk::Boolean m_DesenharHachuraX (AcGiWorldDraw* wd);	Conjunto de 9 funções responsáveis por construir os tipos de hachura requeridos, dependendo da posição do pilar no pavimento superior em relação ao pavimento inferior[**Figura 4-5**]

Tabela 4-4 - Métodos Próprios da Classe CCorte

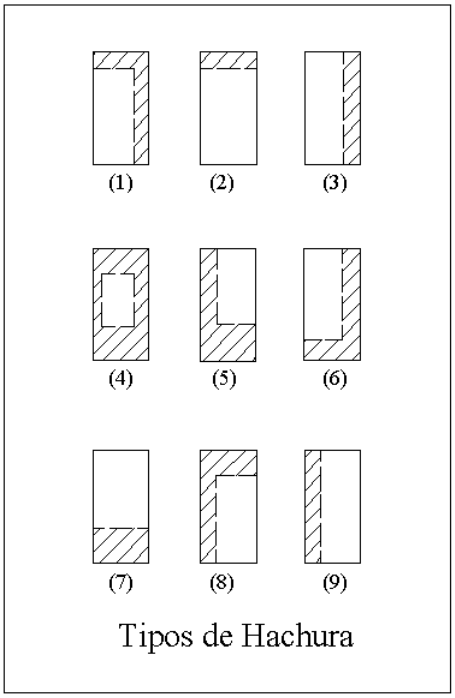


Figura 4-5

Para desenhar os diversos tipos de hachura na transição de pilares, foi necessário desenvolver uma estrutura de código que foi executado conforme está demonstrado nos fluxogramas [**Figura 4-6**] [**Figura 4-7**] [**Figura 4-8**] [**Figura 4-9**].

As funções virtuais derivadas de AcDbEntity terão as suas implementações específicas para a classe CCorte. Um exemplo significativo é a sobrecarga da função worldraw(). Como já mencionado, esta função é a responsável pela construção da representação gráfica da entidade no AutoCAD. Com o auxílio da variável inteira m_iTipoCorte, pode-se determinar , utilizando o comando da linguagem C++ *switch*, qual o tipo de corte a ser desenhado.

```
Adesk::Boolean
CCorte::worldDraw(AcGiWorldDraw* wd)
{
      Acad::ErrorStatus es = Acad::eOk;
      switch(m_iTipoCorte)
      {
            case 1:
                  m_DesenhaCorteViga(wd);
                  break;
            case 2:
                  m_DesenhaCortePilar(wd);
                  break;
      }
      return es;
}
```

A implementação das outras funções virtuais, também deve ser compatível com a entidade costumizada a ser criada, e pode ser encontrada no código fonte do aplicativo.

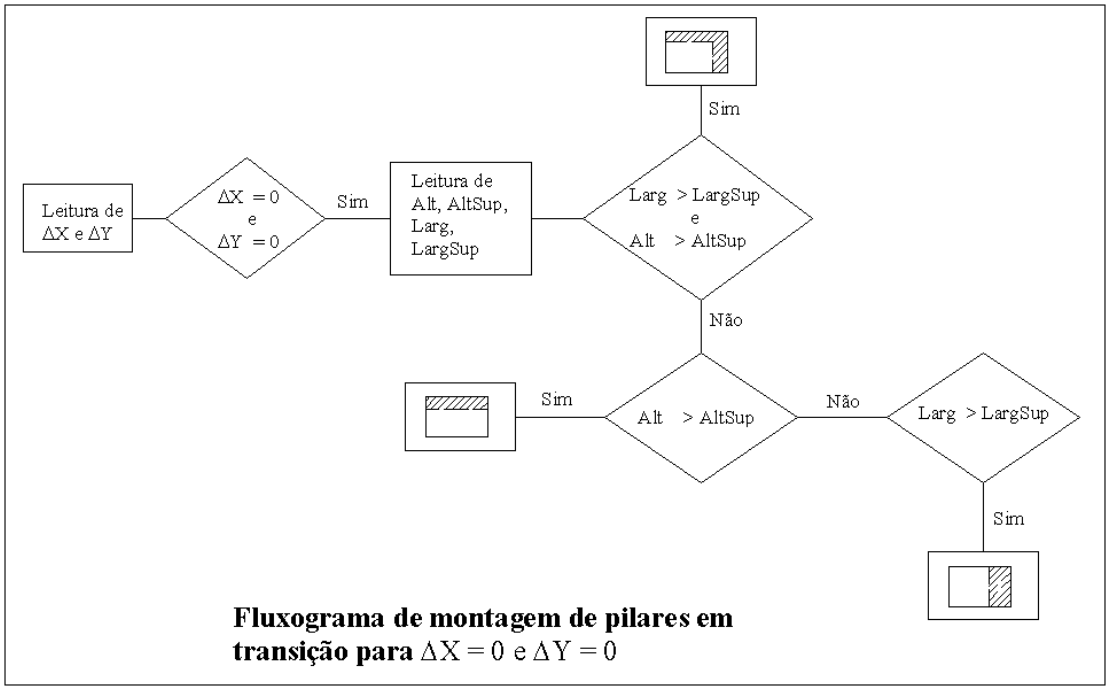


Figura 4-6

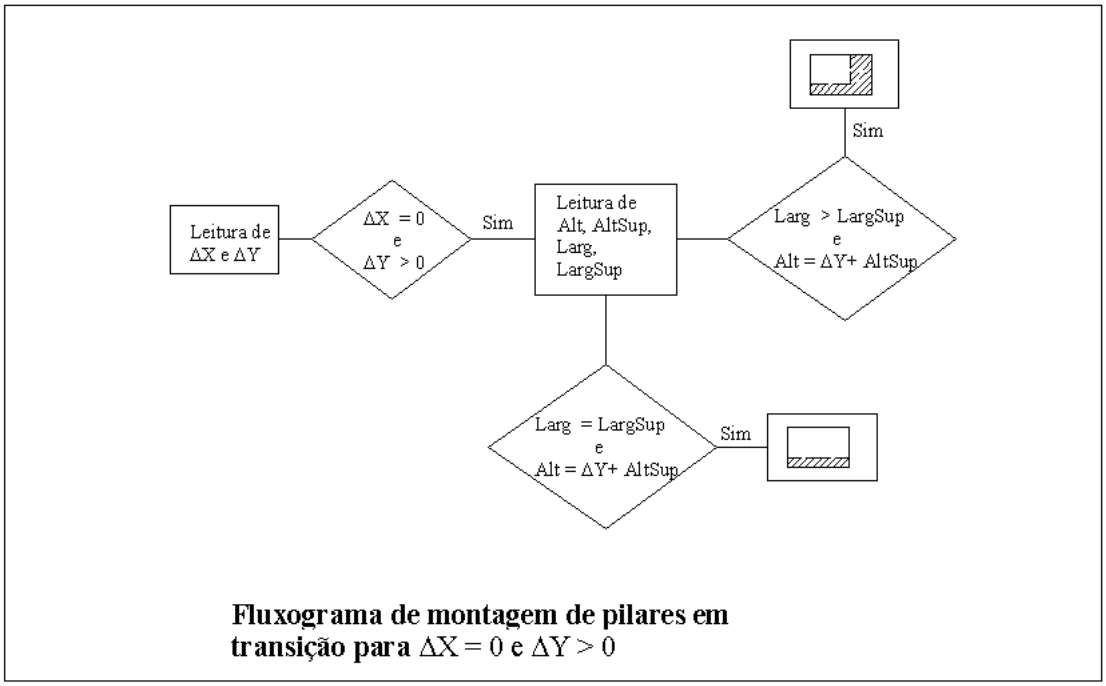
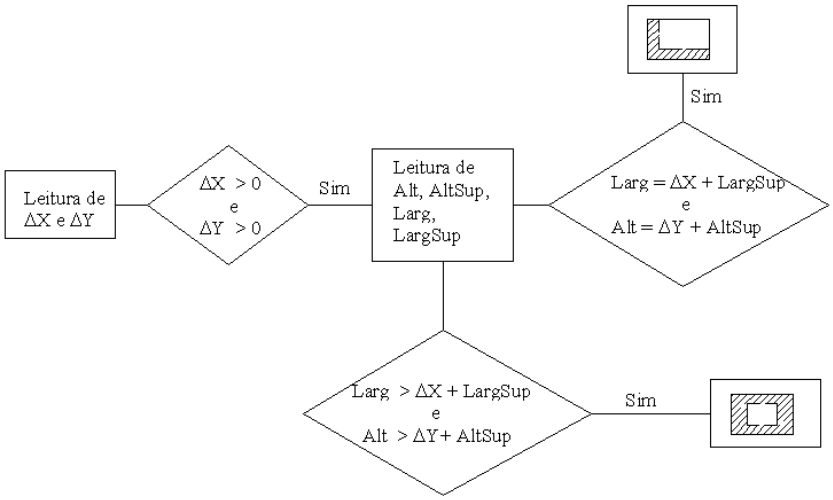
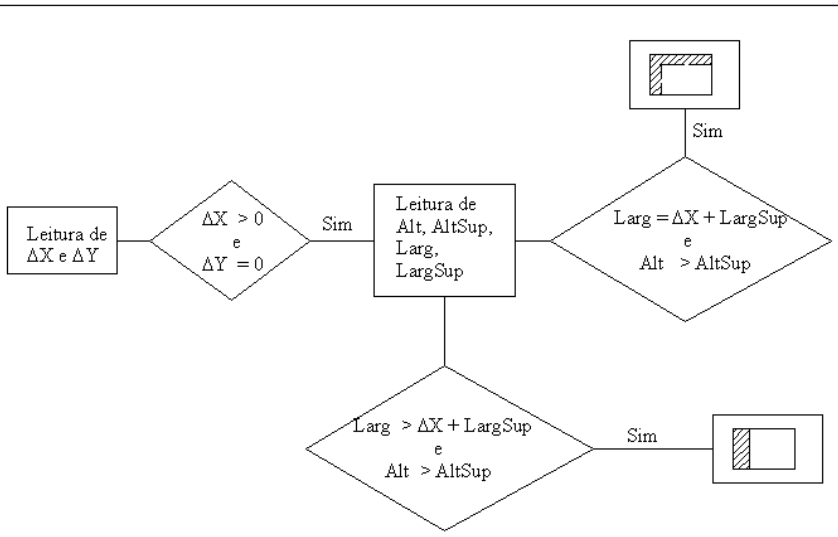


Figura 4-7



Fluxograma de montagem de pilares em transição para $\Delta X > 0$ e $\Delta Y > 0$

Figura 4-8



Fluxograma de montagem de pilares em transição para $\Delta X > 0$ e $\Delta Y = 0$

Figura 4-9

4.2.3 A classe CVista

A classe CVista, de acordo com o mapa de classes do aplicativo [**Figura 3-7**], será derivada da classe CCaixa. Esta Classe&Objeto, é responsável pela representação gráfica das vigas em Vista.

Uma viga pode possuir um ou mais tramos. Ao produzir um desenho de forma costumizada, deve-se observar se existem procedimentos equivalentes, que podem ser repetidos. Em uma Viga todos os tramos possuem parâmetros comuns e são igualmente representados graficamente, exceto o valor dos parâmetros adotado em cada tramo. Portanto, todos os procedimentos adotados para uma viga com apenas um tramo, podem ser igualmente adotado para vigas com mais de um tramo, se forem feitas algumas pequenas modificações. Na primeira versão deste aplicativo, optou-se por dar ênfase à construção de vigas com um tramo e seus diversos elementos, e numa segunda etapa, poder representar vigas com múltiplos vãos.

O desenho de uma Viga em Vista requer parâmetros diversos daqueles já declarados na classe CCaixa tais como: **Largura**, **Altura**, **Largura do Apoio Esquerdo**, **Vão**, **Largura do Apoio Direito**. Agora, os apoios estarão visíveis, representados em Vista seguindo o desenho de fôrma correspondente.

Muitos são os elementos estruturais que podem servir de apoio para as vigas: elementos de fundação (blocos), paredes estruturais ou muros de arrimo, pilares e vigas. Se viga a ser armada encontra-se em balanço ou engastada, ela não terá apoio em uma de suas extremidades.[**Figura 4-10**] Os apoios de tipo Pilar e Viga serão representados nesta primeira versão, além da extremidade em balanço de uma viga engastada. A identificação do tipo de extremidade à direita e à esquerda, se fará através de duas variáveis inteiras que assumirão o valor de 1 a 3.

Analisando o apoio do Tipo Pilar, pode-se notar que ele se apresenta de duas formas distintas: Pilar que Morre e Pilar que Continua.[**Figura 4-10**] Como no caso anterior, aqui também variáveis inteiras assumirão o valor 1 ou 2, guardando a informação do **Tipo de Pilar** à esquerda e\ou à direita da viga bi- apoiada.

No caso do apoio do Tipo Viga, várias representações serão geradas dependendo da **Altura** da Viga de **Apoio** e da sua **Posição** em relação a viga a ser armada.[**Figura 4-11**]

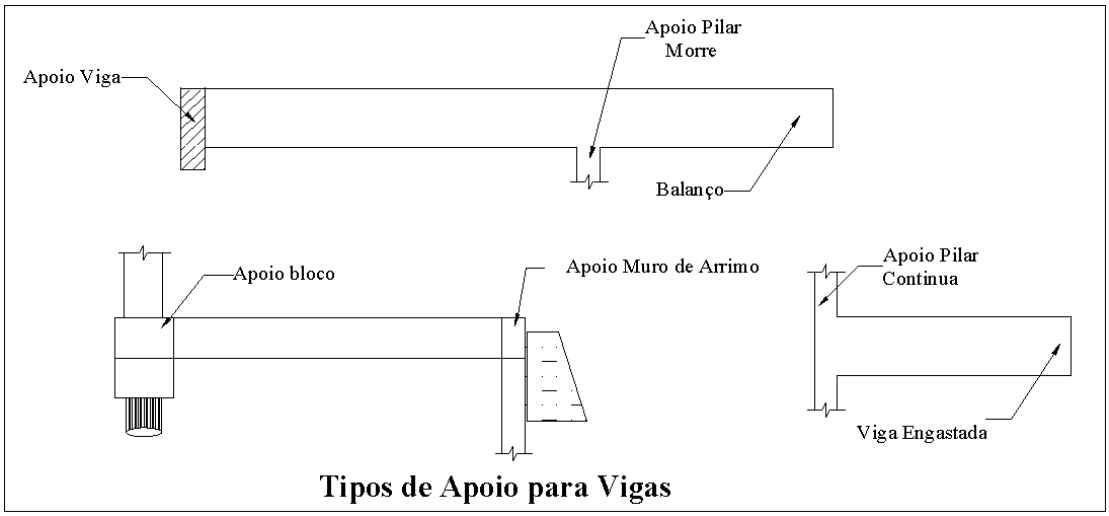


Figura 4-10

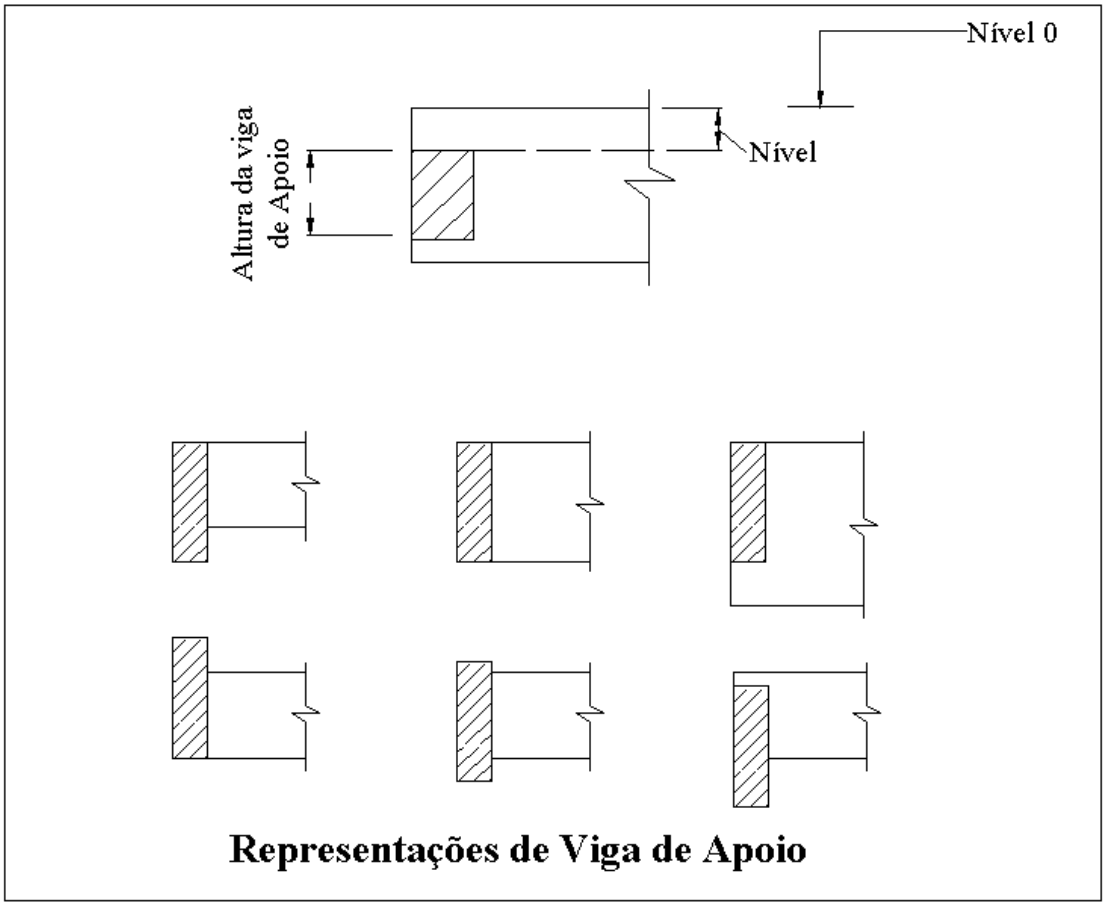


Figura 4-11

A **Tabela 4-5** traz os atributos encapsulados na classe CVista.

Nome do atributo	**Descrição do que representa**
m_iTipoApoioE;	Define qual o tipo de extremidade escolhido para o lado esquerdo da viga. Os valores assumidos serão (1) -Pilar, (2)-Viga e (3)-Balanço.
m_iTipoApoioD	Define qual o tipo de extremidade escolhido para o lado direito da viga. Os valores assumidos serão (1) -Pilar, (2)-Viga e (3)-Balanço.
m_iTipoPilarE	Define qual o tipo de pilar em que está apoiada a extremidade esquerda da viga A variável assumirá o valor (1) -Pilar que Morre e (2) - Pilar que Continua.
m_iTipoPilarD	Define qual o tipo de pilar em que está apoiada a extremidade direita da viga A variável assumirá o valor (1) -Pilar que Morre e (2) - Pilar que Continua.
m_dAltApoioVigaE	Altura da viga de apoio na extremidade esquerda
m_dAltApoioVigaD	Altura da viga de apoio na extremidade direita
m_dNivelE	Distância entre o topo da viga a ser armada e o topo da viga de apoio na extremidade esquerda
m_dNivelD	Distância entre o topo da viga a ser armada e o topo da viga de apoio na extremidade direita
*pHatch	Ponteiro para um objeto da classe AcDbHatch, responsável por capturar os parâmetros necessários para desenhar as hachuras.

Tabela 4-5 - Atributos da classe CVista

Os métodos próprios que ajudarão a construir a representação gráfica da Classe CVista serão:

Nome da Função Membro	**Descrição**
Acad::ErrorStatus m_GetVertices*XXXX* (AcGePoint3dArray& cordVertices)const	Conjunto de 2 funções que constroem o "array" das coordenadas dos vértices das linhas que irão representar o Vão da viga a ser armada [**Figura 4-12**]
Acad::ErrorStatus m_GetVert*YYYY* (AcGePoint3dArray& cordVert)const	Conjunto de 27 funções que constroem array de coordenadas dos pontos para a confecção do desenho dos apoios à esquerda e à direita da viga a ser armada, bem como os vértices das linhas auxiliares, que comporão este desenho.[**Figura 4-12**] e [**Figura 4-13**]
Adesk::Boolean m_DesenhaPilar*XXXX* (AcGiWorldDraw* wd)	Conjunto de 4 funções que constrói, através do ponteiro **wd*, os objetos que representam o apoio de tipo Pilar à esquerda e à direita da viga a ser armada.
Adesk::Boolean m_DesenhaBalanco*HHH* (AcGiWorldDraw* wd)	Conjunto de 2 funções responsáveis pela construção da representação gráfica da extremidade em balanço, à esquerda e à direita da viga a ser armada.
Adesk::Boolean m_DesenhaViga*ZZZ* (AcGiWorldDraw* wd)	Conjunto de 2 funções responsáveis pela construção do apoio do tipo Viga, à esquerda e à direita da viga a ser armada.
Adesk::Boolean m_DesenharHachuraViga*X* (AcGiWorldDraw* wd);	Conjunto de 2 funções responsáveis por construir os tipos de hachura requeridos pela Viga de apoio.

Tabela 4-6 - Métodos Próprios da Classe CVista

Um diagrama de montagem teve que ser confeccionado, para que não houvesse superposição de linhas. [**Figura 4-14**]

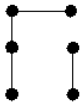
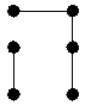
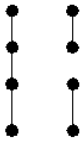
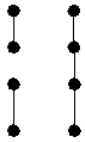
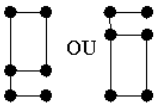
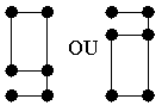
Função	Desenho	Função	Desenho
m_GetVertices1 m_GetVertices2			
m_GetVertPilarMorreE1 m_GetVertPilarMorreE2		m_GetVertPilarMorreD1 m_GetVertPilarMorreD2	
m_GetVertPilarContinuaE1 m_GetVertPilarContinuaE2 m_GetVertPilarContinuaE3		m_GetVertPilarContinuaD1 m_GetVertPilarContinuaD2 m_GetVertPilarContinuaD3	
m_GetVertBalançoE		m_GetVertBalançoD	
m_GetVertVigaE m_GetVertVigaE1 OU m_GetVertVigaE2		m_GetVertVigaD m_GetVertVigaD1 OU m_GetVertVigaD2	
Funções que retornam coordenadas dos vértices - Vão e Apoios			

Figura 4-12

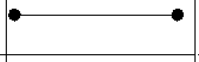
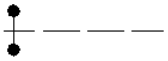
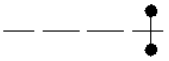
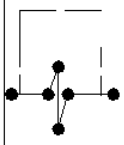
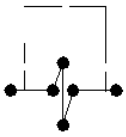
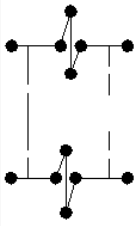
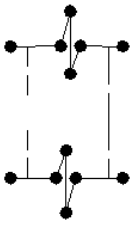
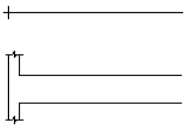
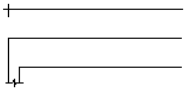
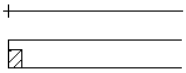
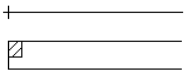
Função	Desenho	Função	Desenho
m_GetVert LinhaChamada			
m_GetVertLinhaChamadaE		m_GetVertLinhaChamadaD	
m_GetVertLinha InterropPME		m_GetVertLinha InterropPMD	
m_GetVertLinha InterropPCE1 m_GetVertLinha InterropPCE2		m_GetVertLinha InterropPCD1 m_GetVertLinha InterropPCD2	
Funções que retornam coordenadas dos vértices - Linhas Auxiliares			

Figura 4-13

Função	Desenho	Função	Desenho
m_DesenhaPilarContinuaE		m_DesenhaPilarContinuaD	
m_DesenhaPilarMorreE		m_DesenhaPilarMorreD	
m_DesenhaBalançoE		m_DesenhaBalançoD	

m_DesenhaVigaE	if (m_dNivelE < 0)
	 
	if (m_dAlt > m_dAltApoioVigaE - m_dNivelE)
	 
		m_DesenhaVigaD	if (m_dNivelE < 0)
			 
			if (m_dAlt > m_dAltApoioVigaE - m_dNivelE)
			 

Funções que desenham a Viga em Vista			

Figura 4-14

4.3 O Módulo Armação:

(Descrição do módulo Armação, destacando os membros de cada classe derivada)

4.3.1 A classe CArmacaoVista

No mapa de classes, a classe CArmacaoVista é derivada de CVista, e conseqüentemente de CCaixa. [**Figura 3-7**] Além dos atributos já declarados, deve-se abstrair outros, retirados do modelo representativo de uma armação em vista.[**Figura 4-15**]

Qualquer barra da armadura, inclusive de distribuição, de montagem e os estribos, deve ter o **Cobrimento** de concreto pelo menos igual ao seu diâmetro mas nunca menor que valores determinados pela norma, dependendo do local onde ficará a peça detalhada (concreto revestido, aparente, em contato com o solo ou em meio fortemente agressivo) [**NBR 6118**]

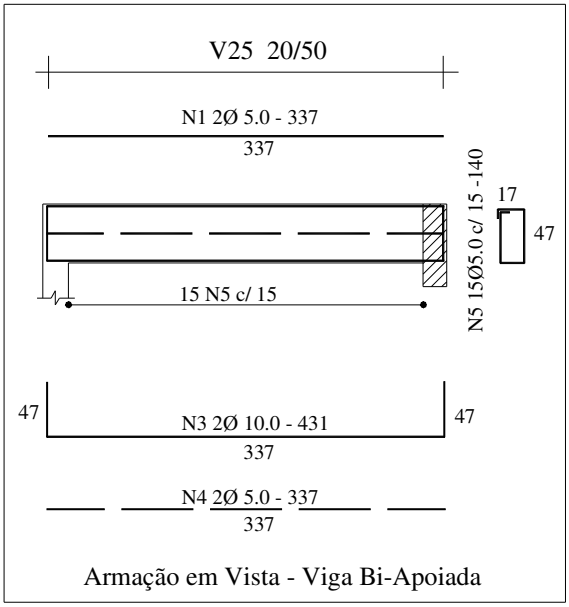


Figura 4-15

No detalhamento de peças em concreto armado, a barra será responsável por combater os esforços aplicados à estrutura. Uma forma de definir quais serão os elementos componentes da armadura de uma viga, é analisando os seus diagramas de momento fletor, força cortante [**Figura 4-16**]. Eles servem de base para a determinação do comprimento das barras.

Agora se pode imaginar qual tipo de armadura que deverá "cobrir" estes diagramas. No caso das vigas bi-apoiadas, como o momento fletor é positivo a armadura superior será sempre reta. [**Figura 4-17**] Esta armadura pode ser considerada uma armadura de montagem, sem valor estrutural ou dependendo do valor do momento fletor e das dimensões da peça, pode se tornar uma armadura de compressão (armadura dupla). Para armadura inferior, é necessário analisar o diagrama de momento fletor e executar o detalhamento observando o comprimento de ancoragem necessário dependendo da bitola utilizada.

Para o momento positivo, a zona de ancoragem começa a partir do apoio. Porém, somente a largura do apoio não é suficiente e a ferragem terá que sofrer uma dobra a 90 graus até alcançar o valor da ancoragem requerido pela norma. Na maioria das vezes, se a ferragem se estender ao longo da altura até alcançar o topo da viga, a barra já estará ancorada satisfatoriamente. De acordo com as normas construtivas, pelo menos 2 barras deverão sofrer dobra a 90º, facilitando a confecção da grade de barras. Esta armadura será chamada de Principal. Outras barras poderão ser usadas

para completar a área de ferro necessária podem ser retas ou dobradas a 60º ou 45º, caso ajudem a combater também o cisalhamento. Aqui elas serão chamadas de Armaduras Secundárias.

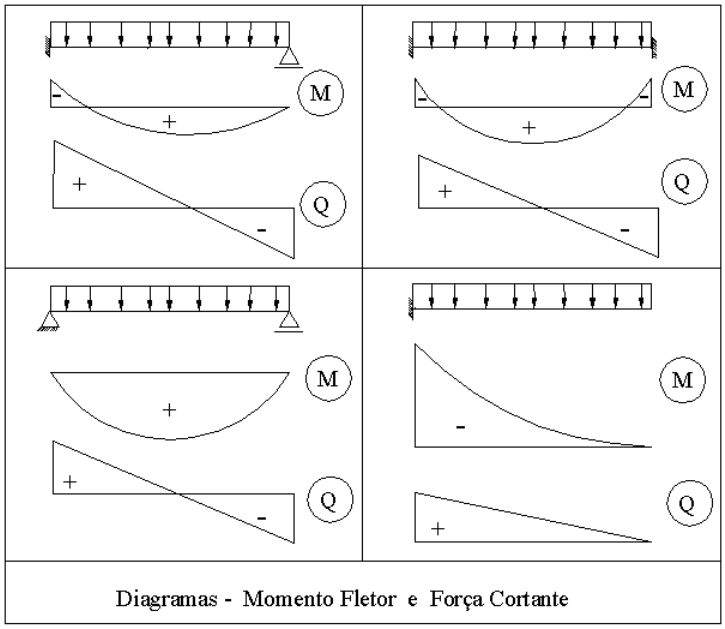


Figura 4-16

As armaduras secundárias, também deverão ter o seu respectivo comprimento de ancoragem, mas neste caso, a zona de ancoragem não começa no apoio, e dependerá do formato do diagrama de momento fletor e da quantidade de barras principais adotada.

Para armar vigas bi-apoiadas este aplicativo disporá da armadura positiva principal, dobrada a 90º e da armadura positiva secundária do tipo reta, que se estenderá até o final do apoio à esquerda e à direita. Caberá ao usuário utilizar apenas a armadura principal ou conjugar as duas, de forma a cobrir, da melhor maneira possível, o diagrama de momento fletor. [**Figura 4-17**]

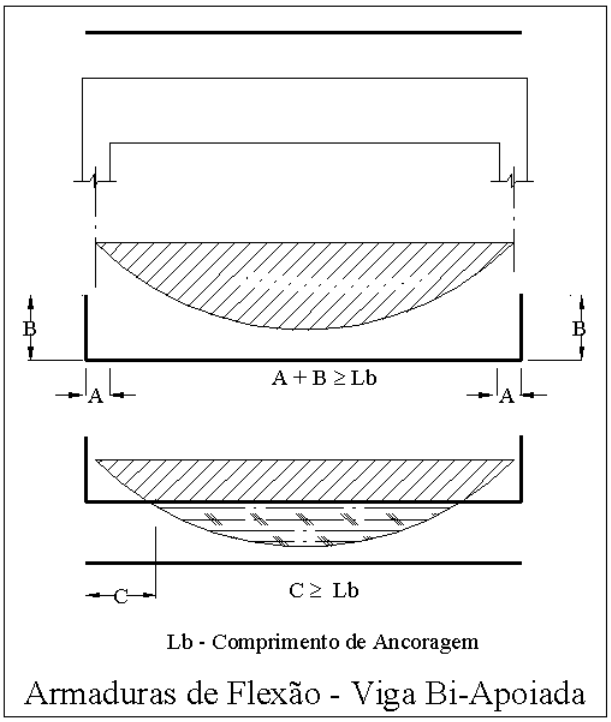


Figura 4-17

Analisando o diagrama de esforços para a viga bi-engastada, nota-se o aparecimento de momentos fletores negativos. A zona de ancoragem de sua armadura negativa não começará nos apoios. Para ancorar as barras de forma adequada, o procedimento será dobrar a armadura a 90º e nas duas extremidades prolongá-la, ao longo da altura da viga, até a sua base. Este será o comprimento de ancoragem para as barras que irão combater o momento negativo. Os esforços decorrentes do momento positivo serão combatidos como na viga bi-apoiada.[**Figura 4-18**].

Uma viga que possui uma de suas extremidades engastadas e a outra livre, só possui momento negativo no apoio engastado. A armadura principal que irá combater este tipo de esforço deverá ser dobrada a 90º, na extremidade engastada e reta na extremidade livre.[**Figura 4-19**]

Um exemplo de armadura para combater os momentos fletores em uma viga engastada e apoiada está descrito na **Figura 4-20**.

Durante muitos anos, os engenheiros calculistas utilizaram a armadura dobrada a 60º e a 45º, para ajudar a combater o cisalhamento. Neste aplicativo, o cisalhamento só será combatido com a utilização de estribos.

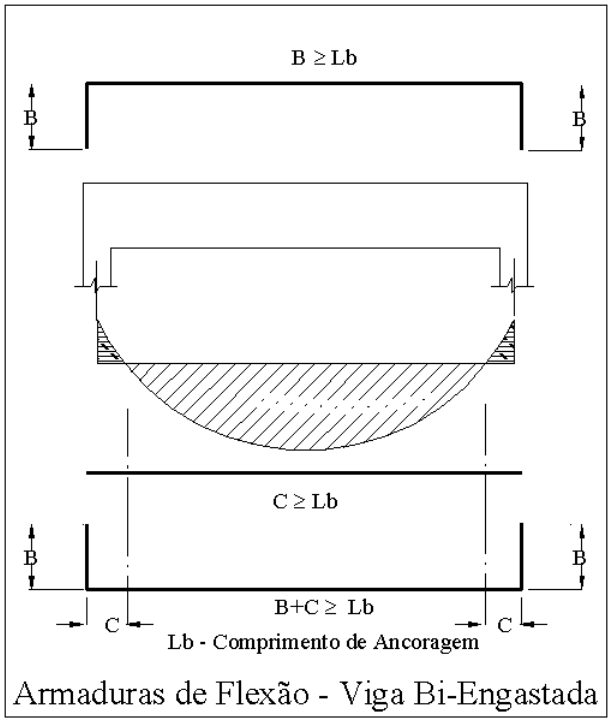


Figura 4-18

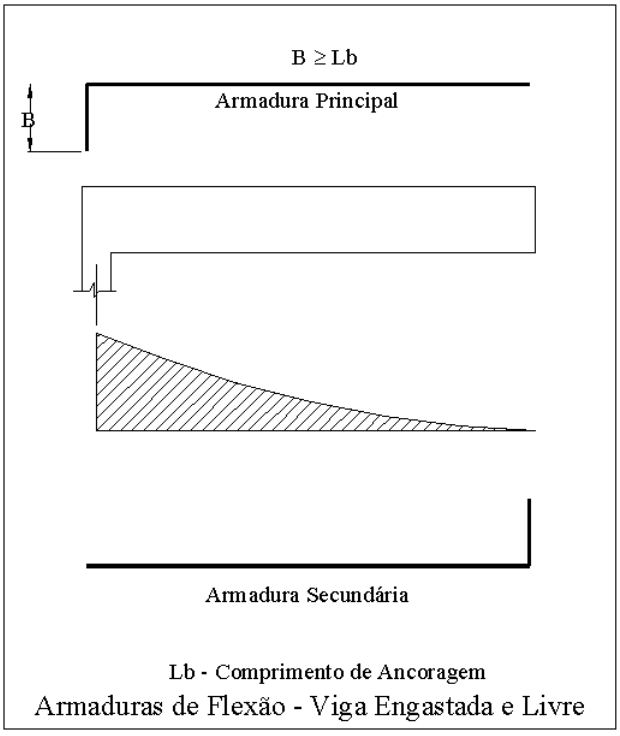


Figura 4-19

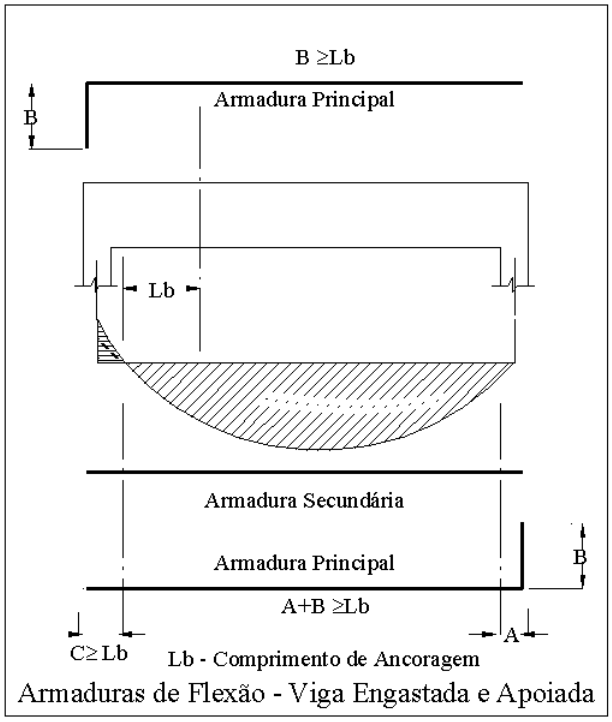


Figura 4-20

A armadura do tipo Costela serve para combater os esforços de fissuração, decorrentes da retração e variações térmicas a que estão sujeitas as vigas com altura maior que 60 cm. Neste aplicativo, este tipo de armadura poderá ser utilizado.

Como demonstrado, são vários tipos de barras disponíveis e para identificá-las foi necessário criar um **"Array de Booleanos"** que irá indicar quais os tipos de armadura foram selecionados pelo usuário.

Para garantir uma boa concretagem, sem ninhos (vazios) a NBR 6118 estabeleceu um espaçamento mínimo entre as barras. Portanto, haverá um número máximo de barras suportado pela largura de uma viga. Muitas vezes, a quantidade de barras é superior a capacidade da seção e é necessário utilizar uma segunda camada. Para comunicar ao programa se existirá ou não esta segunda camada, foi criada a variável booleana *m_iSegundaCamada*.

A **Área da Armadura** a ser detalhada será um dado de entrada. Este dado deverá ser fornecido separadamente para os seguintes tipos de armadura: Armadura Principal Superior, Armadura Secundária Superior, Armadura Principal Inferior, Armadura Secundária Inferior, Estribo e Costela. Com esta informação, o programa constroí uma lista de strings, onde o usuário poderá escolher qual o **Diâmetro**, a **Quantidade** de barras. A explicação de como se dará a construção desta lista poderá

ser encontrada no Capítulo 6. Depois da escolha, estas informações são separadas em atributos e inseridas do banco de dados, facilitando a construção da Lista de Materiais.

A seguir, a tabela de Atributos da classe CArmacaoVista:

Nome do atributo	**Descrição do que representa**
m_dCobrimento	Armazena a distância do cobrimento (distância entre a caixa de madeira e a armadura mais superficial) adotada para a Viga que está sendo armada.
m_iTipoArm[ ]	Array boleanos que identifica quais os tipos de armadura foram selecionadas pelo usuário.
m_iSegundaCamada[ ]	Array de boleanos que identifica se existe segunda camada para os tipos de armadura que a suportam (Armadura Secundária Inferior e Armadura Secundária Superior).
m_dArea*XXXX*	Armazena a área de armadura necessária para cada um dos tipos de armadura disponíveis.
m_csArma*YYYY*Selec	Armazena a string selecionada pelo usuário, correspondente a bitola e quantidade de barras para cada tipo de armadura selecionada.
m_dQuantBitola*ZZZZ*	Armazena a quantidade de barras selecionada pelo usuário para cada tipos de armadura.
m_dDiametro*XXXX*	Armazena o diâmetro selecionado pelo usuário para cada tipo de armadura.

Tabela 4-7 Atributos da classe CArmacaoVista

Os métodos que ajudarão a construir a representação gráfica da armadura de uma viga em vista serão:

Nome da Função Membro	**Descrição**
Acad::ErrorStatus m_GetVert*XXXX*(AcGePoint3dArray& cordVertices)const	Conjunto de 21 funções responsáveis por construir "arrays" de coordenadas dos pontos das diversas representações gráficas de armaduras disponíveis no aplicativo. Tais desenhos serão inseridos em seus locais apropriados no desenho, dentro e fora da Caixa, e estarão subordinados ao **Ponto de Inserção** definido pelo usuário, quando da construção da Caixa da Viga em Vista..
Adesk::Boolean m_Desenha*YYYY*(AcGiWorldDraw* wd)	Conjunto 10 funções responsáveis por construir a representação gráfica dos diversos tipos de armadura que foram selecionadas pelo usuário, com seu respectivo texto, layer e linhas auxiliares. Através da montagem correta destes desenhos será formado o modelo desejado da armadura da viga.

Tabela 4-8 - Métodos próprios da classe CArmacaoVista

4.3.2 A classe CArmacaoCorte

A classe CArmacaoCorte é derivada da classe CCorte e portanto herdará seus atributos e métodos, valiosos para a construção da representação das armaduras. [**Figura 3-7**]. Observando com atenção pode-se perceber que esta classe tem várias semelhanças com a classe CArmacaoVista, quando se pensa na representação gráfica da armadura. Porém, além de ser representada em Vista, agora a armadura será visualizada também em Corte. Cada barra será apresentada devidamente posicionada dentro da caixa, com sua seção transversal circular. [**Figura 4-21**]. Só serão armadas em corte as vigas bi-apoiadas e bi-engastadas. Isto se deve ao fato de que a utilização

de extremidades diferentes pode gerar dúvida na hora da execução, pois os apoios não estão visíveis no desenho em corte.

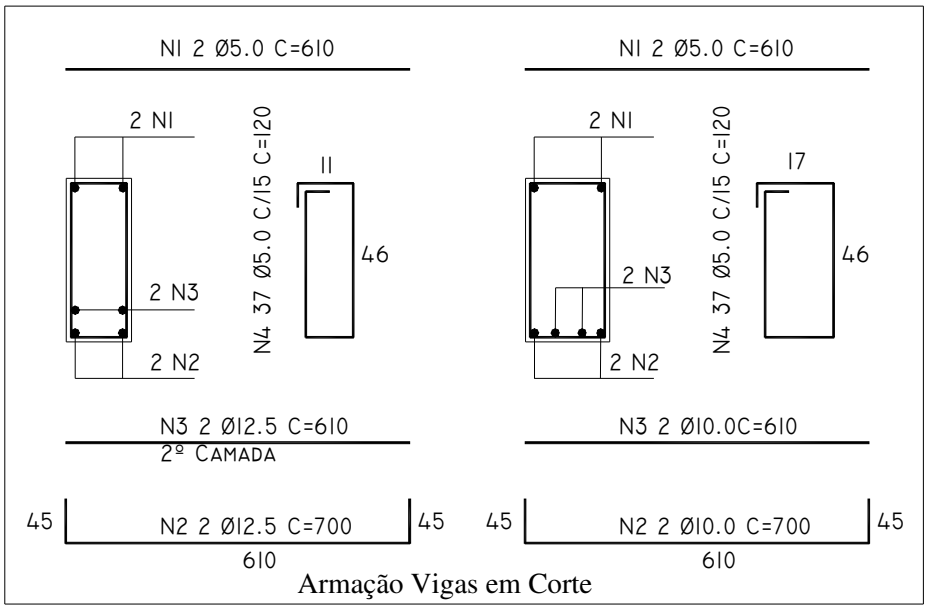


Figura 4-21

Como já discutido no item 4-3-1, a Armadura Secundária pode estar localizada na primeira ou na segunda camada. Cada barra desenhada em corte deverá ser identificada facilitando a determinação da sua localização dentro da caixa. Quando a armadura principal e a armadura secundária estão na primeira camada, é necessário criar um mecanismo que faça a identificação de qual é a principal e qual é a secundária. Por questões construtivas as armaduras posicionadas na extremidade da seção transversal (*N2* –**Figura 4-21**) devem ser do tipo Principal. As outras barras têm posição livre sendo aconselhável manter a simetria. Depois de posicionar as barras principais e secundárias em corte transversal dentro da caixa, será a linha de chamada é que fará a sua identificação.

Os atributos da classe CArmacaoCorte são:

Nome do atributo	**Descrição do que representa**
m_dCobrimento	Armazena a distância do cobrimento (distância entre a caixa de madeira e a armadura mais superficial) adotada para a Viga que está sendo armada.
m_iTipoArm[ ]	Array boleanos que identifica quais os tipos de armadura foram selecionadas pelo usuário.
m_iSegundaCamada[ ]	Array de boleanos que identifica se existe segunda camada para os tipos de armadura que a suportam (Armadura Secundária Inferior e Armadura Secundária Superior).
m_dArea*XXXX*	Armazena a área de armadura fornecida pelo usuário para cada um dos tipos de armadura.
m_csArma*YYYY*Selec	Armazena a string selecionada pelo usuário, correspondente a bitola e quantidade de barras para cada tipo de armadura.
m_dQuantBitola*ZZZZ*	Armazena a quantidade de barras selecionada pelo usuário para cada tipo de armadura
m_dDiametro*XXXX*	Armazena o diâmetro selecionado pelo usuário para cada tipo de armadura.

Tabela 4-9 Atributos da classe CArmacaoCorte

Os métodos que ajudarão a construir a representação gráfica da armadura de uma viga em corte serão:

Nome da Função Membro	Descrição
Acad::ErrorStatus m_GetVert*XXXX*(AcGePoint3dArray& cordVertices)const	Conjunto de 7 funções que constroem "arrays" de coordenadas dos pontos para desenhar as diversas armaduras disponíveis. Tais desenhos serão inseridos em seus locais apropriados fora da Caixa, e estarão subordinados ao **Ponto de Inserção** definido pelo usuário, quando da construção da Caixa da Viga em Corte.
Acad::ErrorStatus m_GetCentroBitola*YYYY*(double dQuantBitolas,	Conjunto de 7 funções que constroem "arrays" de coordenadas dos pontos para desenhar das diversas armaduras diponíveis. Tais desenhos serão inseridos em seus locais apropriados dentro da Caixa, e estarão subordinados ao **Ponto de Inserção** definido pelo usuário, quando da construção da Caixa da Viga em Corte
Acad::ErrorStatus m_GetVertLinhaChama*ZZZZ* (double dQuantBitolas,AcGePoint3dArray& cordVertices)const	Conjunto de 4 funções que constroem "arrays" de coordenadas dos pontos que irão determinar posição das linhas de chamada das barras representadas em corte dentro da caixa e estarão subordinados ao **Ponto de Inserção** definido pelo usuário, quando da construção da Caixa da Viga em Corte
Adesk::Boolean m_Desenha*YYYY*(AcGiWorldDraw* wd)	Conjunto 6 funções responsáveis construir a representação gráfica dos diversos tipos de armadura com seu respectivo texto, layer e linhas auxiliares. Através da montagem correta destes desenhos será construído o modelo de armação da viga em corte.

Tabela 4-10 - Métodos próprios da classe CArmacaoCorte

4.3.3 A classe CArmacaoPilar

Inicialmente, pensou-se em não criar esta classe, construindo dois objetos na classe CArmacaoCorte que representariam a armação de Vigas e Pilares em corte. Mas, o desenho da armadura de um pilar não possui nenhuma característica comum com o desenho da armadura de uma viga.

Seguindo o critério da Modularidade, importante condição para otimizar o processo de desenvolvimento, foi criada a classe CArmacaoPilar. Cada módulo poderá ser compilado em separado facilitando a fase de testes. Isto permitirá o trabalho em equipe, pois cada programador ficará responsável por um módulo.

A classe CArmacaoPilar será derivada de CCorte, pois ela precisa das informações de fôrma armazenadas neste módulo. Com estas e outras informações próprias da representação da armadura de Pilares, pode-se criar o modelo de armação pretendido.

A sistemática de criação de função para armazenar cada um dos tipos de barra encontradas em um desenho de Pilar [**Figura 3-5**] será usada, a exemplo das classes anteriores, através da criação de "Arrays de Booleanos" que irão armazenar os tipos de barras selecionados pelo usuário.

Quanto aos atributos, eles também seguirão a forma adotada pelas classes CArmacaoCorte e CArmacaoVista.

Como acontece nas vigas representadas em corte, as barras que constituem a armação longitudinal de um Pilar estarão representadas em vista, fora da caixa e, em corte, posicionadas dentro da caixa. Podem existir três tipos de barras: Armadura Morre, Armadura Continua, Armadura Espera. [**Figura 3-5**] O usuário deverá fornecer a área necessária para combater os esforços no pavimento inferior e no pavimento superior, escolhendo a bitola e a quantidade de barras, que lhe serão apresentadas na lista de strings que aparece no quadro de diálogos.

Para representar a armadura do Pilar dentro da caixa em corte, basta que se tenha a informação de quantas barras de cada tipo (Morre, Continua e Nasce) serão necessárias. Para fazer esta determinação serão utilizados os modelos de transição de pilares construídos na classe CCorte.

Cada tipo de barra possui uma representação gráfica distinta [**Figura 3-3**] tanto em corte quanto vista. Portanto é necessário criar funções distintas, que construam estes desenhos.

A **Tabela 4-11** mostra os atributos encapsulados na classe CArmacaoPilar

Nome do atributo	**Descrição do que representa**
m_dCobrimento	Armazena a distância do cobrimento (distância entre a caixa de madeira e a armadura mais superficial) adotada para o Pilar que está sendo armado.
m_iTipoArm[ ]	"Array de boleanos" que identifica quais os tipos de armadura que foi selecionado pelo usuário(Armadura Pavimento Superior, Armadura Pavimento Inferior, Estribos, Espaçadores)
m_dArea*XXXX*	Armazena a área de armadura fornecida pelo usuário para cada tipo disponível (Armadura Pavimento Superior, Armadura Pavimento Inferior, Estribos, Espaçadores)
m_csArma*YYYY*Selec	Armazena a string selecionada pelo usuário, correspondente a bitola e quantidade de barras para cada tipo de armadura selecionada.
m_dQuantBitola*ZZZZ*	Armazena a quantidade de barras selecionada pelo usuário em cada tipo de armadura disponível
m_dDiametro*XXXX*	Armazena o diâmetro selecionado pelo usuário para cada tipo de armadura disponível

Tabela 4-11 Atributos da classe CArmacaoPilar

Os métodos que ajudarão a construir a representação gráfica da armadura de um pilar em corte serão:

Nome da Função Membro	**Descrição**
Acad::ErrorStatus m_GetVert*XXXX*(AcGePoint3dArray& cordVertices)const	Conjunto que constroem "arrays" de coordenadas dos pontos para desenhar as diversas armaduras que aparecem em vista. (Armadura Morre, Armadura Continua, Armadura Nasce, Estribos e Espaçadores)
Acad::ErrorStatus m_GetCentroBitola*YYYY*(double dQuantBitolas,	Conjunto de 7 funções que constroem "arrays" de coordenadas do centro das circunferências das barras longitudinais representadas em corte dentro da caixa.
Acad::ErrorStatus m_GetVertLinhaChama*ZZZZ* (double dQuantBitolas,AcGePoint3dArray& cordVertices)const	Conjunto de funções que constroem "arrays" de coordenadas dos pontos das linhas de chamada existentes em um desenho de armação de pilares, quando da construção da Caixa da Viga em Corte
Adesk::Boolean m_Desenha*YYYY*(AcGiWorldDraw* wd)	Conjunto de funções responsáveis construir a representação gráfica dos diversos tipos de armadura em corte e em vista com seu respectivo texto, layer e linhas auxiliares. Através da montagem correta destes desenhos será construído o modelo de armação da viga em corte.

Tabela 4-12 - Métodos próprios da classe CArmacaoPilar

4.4 O Módulo Lista:

(Descrição do módulo, destacando os seus atributos e membros encapsulados)

Conforme o mapa de classes do aplicativo [**Figura 3-7**], a classe CLista, é uma classe distinta, que não se apresenta inserida na hierarquia de classes do Aplicativo ArmaCAD. Para capturar os atributos armazenados nas classes CArmacaoCorte, CArmacaoVista e CArmacaoPilar, basta declarar seus objetos dentro de CLista e inseri-los na lista em locais apropriados conforme mostra a **Figura 3-6**.

Cada barra inserida no desenho durante o processo de armação deverá receber um número. Tal número permitirá a sua identificação no desenho de armação e na lista a ser construída.

A maneira mais fácil de capturar os valores que irão compor a lista de materiais é durante a criação das armaduras. No primeiro instante o "array" de números que farão a identificação da barra está vazio. Também estarão vazios os "arrays" que irão armazenar os outros atributos da lista. Quando o usuário criar a armadura da primeira peça, cada barra será numerada e seus atributos serão inseridos nos "arrays" que correspondem a cada coluna da tabela. Agora, como fazer para inserir os valores da segunda peça a ser armada? Basta criar um interador que irá percorrer o "array" existente e descobrir qual o número da próxima barra a ser inserida.

Com isto, independente do tipo de peça que está sendo armada, todas as entidades que fizerem parte deste desenho, terão os seus atributos armazenados e inseridos na lista de ferros.

Além da lista que identifica cada uma das barras que compõem o desenho, uma tabela Resumo também deve ser criada. Mais uma vez, com a ajuda de um interador, esta lista será percorrida identificando quais as barras que possuem a mesma bitola. O comprimento total das diversas barras com a mesma bitola poderão então ser somados e inseridos na tabela resumo. Como a compra da ferragem é feita por quilo, é necessário multiplicar o comprimento total encontrado pelo peso da bitola correspondente. Assim a lista resumo poderá também ser construída e inserida no desenho.

Os atributos da classe CLista serão:

Nome do atributo	**Descrição do que representa**
m_iNumBarra[]	"Array" que irá armazenar o número de identificação da barra
m_iQuantidade[]	"Array" que irá amazenar a quantidade de barras do mesmo tipo existente em uma determinada peça e que possuem a mesma identificação.
m_dDiâmetro[]	"Array" que armazena o diâmetro da barra na ordem em que ela que foi identificada.
m_iCompUnitário[]	"Array" que armazena o comprimento total unitário da barra na ordem em que ela foi identificada.
m_iCompTotal[]	"Array" que armazena o valor resultante da multiplicação do comprimento total pela quantidade de barras existentes na ordem em que ela foi identificada.
m_iCompTotalResumo[]	"Array" que armazena o comprimento total das barras dependendo do diâmetro das bitolas encontradas no desenho.
m_iPesoTotal[]	"Array" que armazena o peso total encontrado para as várias bitola encontradas no desenho.

Tabela 4-13 Atributos da Classe CLista

As funções membros próprias da classe CLista serão:

Nome da Função Membro	**Descrição**
Acad::ErrorStatus m_GetVertLista(AcGePoint3dArray& cordVertices)const	Função que constroi o "array" das coordenadas dos pontos que irão definir o esboço da tabela "Lista de Ferros"
Acad::ErrorStatus m_GetVertListaResumo(AcGePoint3dArray& cordVertices)const	Função que constroi o "array" das coordenadas dos pontos que irão definir o esboço da tabela "Resumo"
Adesk::Boolean m_DesenhaLista(AcGiWorldDraw* wd)	Função que constrói a representação gráfica da tabela "Lista de Ferros" inserindo os "arrays" de atributos correspondentes.
Adesk::Boolean m_DesenhaListaResumo (AcGiWorldDraw* wd)	Função que constrói a representação gráfica da tabela "Resumo" inserindo os "arrays" de atributos correspondentes.

Tabela 4-14 Métodos próprios da classe CLista

Capítulo 6

DESCRIÇÃO DA INTERFACE COM USUÁRIO DO APLICATIVO ARMACAD VERSÃO 1.0

Aspectos comuns ao implementar uma classe derivada de classes do MFC

Interface da classe CCaixa

Interface da classe CVista

Interface da classe CCorte

Interface da classe CArmacaoVista

Interface da classe CArmacaoCorte

Interface da classe CArmacaoPilar

Interface da classe CLista

6.1 Aspectos comuns ao implementar uma classe derivada de MFC

As classes que serão implementadas para servir de interface para este aplicativo, serão derivadas da classe do MFC *CDialog*. Esta classe é usada para exibir caixas de diálogos na tela. Uma caixa de diálogos pode ser de dois tipos: *modal e modeless*. A primeira, deve ser fechada pelo usuário antes que se possa continuar a aplicação. A segunda, permite que o usuário execute outra tarefa sem precisar cancelar ou remover a caixa de diálogos. Neste aplicativo optou-se por usar diálogos de tipo *modal*.

Quando o projeto já está configurado para utilizar a biblioteca MFC, a aba "Resource" aparece janela do "Workspace". Dentro dela, existem vários tipos de pastas. Para criar uma nova caixa de diálogos basta clicar na pasta "Dialog" com o botão direito do mouse escolhendo a opção "Insert New Dialog".Usando o editor de diálogos é possível criar um modelo e depois armazená-lo na pasta “Dialog”.

Uma caixa de diálogos, como qualquer outra janela, recebe mensagens do Windows. Tais caixas devem ser capazes de controlar estas mensagens e ainda controlar a sua interação com o usuário.

Para auxiliar a implementação das classes derivadas de MFC, o Visual C++ 6.0 disponibiliza um mecanismo de ajuda (ClassWizard). O objetivo deste dispositivo é fazer a montagem destas classes, faltando apenas definir a implementação adequada de cada função. Ao desenvolver o aplicativo ArmaCAD este mecanismo de ajuda não utilizado.

A exemplo do que acontece com o ObjectARX aqui também deve-se sobrecarregar funções virtuais herdadas da classe pai *CDialog*. Para algumas delas a sobrecarga é obrigatória:

Nome da Função Sobrecarregada	Descrição
virtual void DoDataExchange (CDataExchange* pDX)	Função responsável por transferir os valores definidos nas caixas de controle para as variáveis da classe derivada que está sendo construída.
virtual BOOL OnInitDialog()	Define o "estado" de cada componente da caixa de diálogos quando esta é inicializada
virtual void OnCancel();	Função responsável pela implementação das ações decorrentes da seleção do botão "Cancel"
virtual void OnOK()	Função responsável pela implementação das ações decorrentes da seleção do botão "OK"

Tabela 6-1 - Alguns métodos da classe CDialog com sobrecarga obrigatória

Para implementar a interface com o usuário, cada Classe&Objeto que foi criada ao desenvolver o aplicativo, terá uma classe derivada de CDialog correspondente. Esta classe será responsável por fornecer mecanismos de entrada de dados intuitivos e de fácil compreensão.

6.2 Interface da classe CCaixa

Apesar de ser uma classe abstrata, a classe CCaixa terá uma interface para colher os dados de projeto necessários a para calcular o comprimento de ancoragem das barras que irão compor a armadura das peças detalhadas.

A caixa de diálogos “Parâmetros de Projeto” será acionada automaticamente logo que o aplicativo for inicializado dentro do AutoCAD. Como estes dados não poderão ser alterados, durante as etapas de criação das entidades, tais membros serão declarados privados (*private*) da classe CCaixa. [**Figura 6-1]**

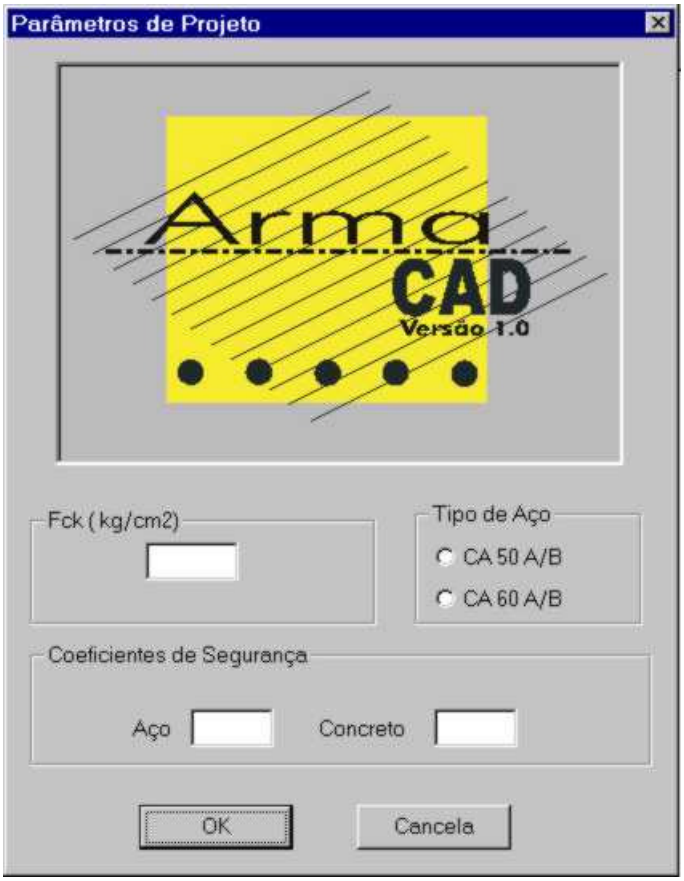


Figura 6-1 – Janela “Parâmetros de Projeto”

6.3 Interface da classe CVista

Os atributos necessários para construir o desenho de uma Viga em Vista, definidos na classe CVista e na classe CCaixa, serão informados pelo usuário, através de uma entrada de dados. A classe CArmacadAPP1, será a responsável por captar estes valores, que depois serão transferidos para as variáveis da classe CVista. [**Figura 6-2**]

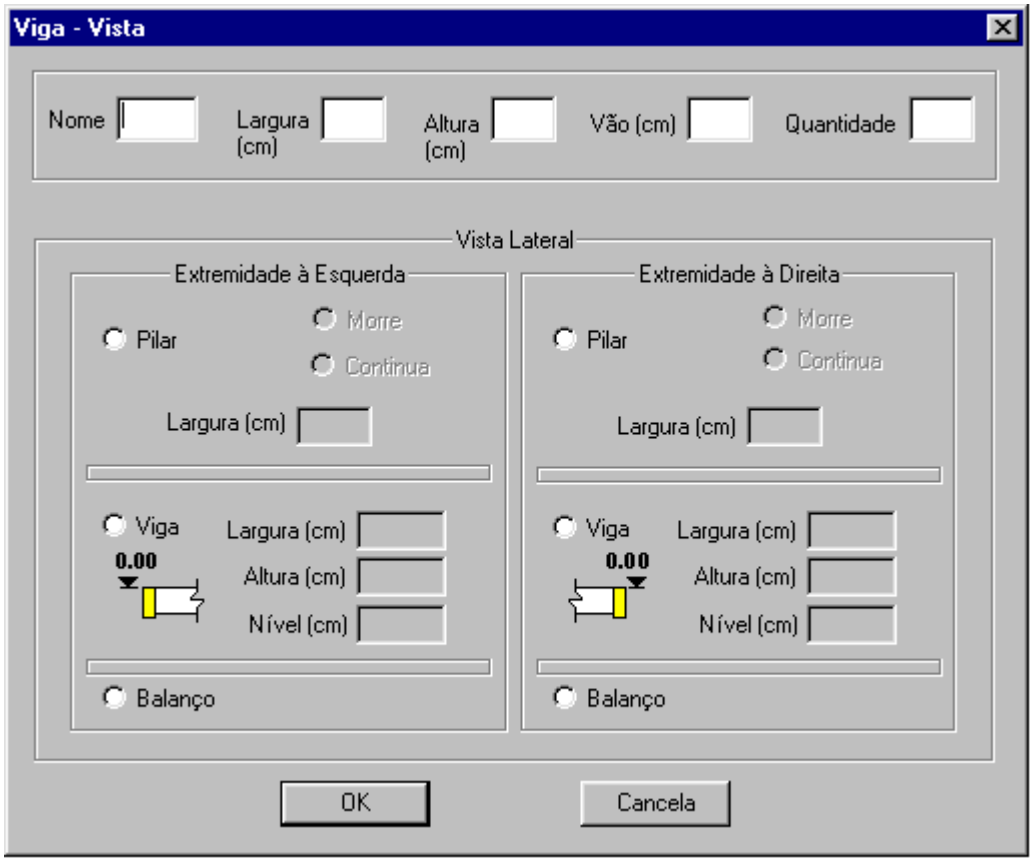


Figura 6-2 - Janela "Viga – Vista"

A **Tabela 6-2** exibe os atributos da classe CArmacadAPP1. Todos eles possuem um correspondente nas classes CVista ou CCaixa:

Nome do atributo	**Descrição do que representa**
m_csNome m_csAltura m_csLargura m_csVao m_csQuantidade m_csLargPilarE m_csLargVigaApoioE m_csAltVigaApoioE m_csNivelE m_csLargPilarD m_csLargVigaApoioD m_csAltVigaApoioD m_csNivelD	Atributos responsáveis por armazenar os dados fornecidos pelo usuário no quadro de diálogos. Tais dados são necessários para construir a representação gráfica da Viga em Vista .
m_iApoioE m_iApoioD	Atributos responsáveis por armazenar valores que irão definir o tipo de apoio selecionado pelo usuário. Eles assumirão o valor (0), (1) ou (2) quando um dos Botões de Rádio "Pilar", "Viga" ou "Balanço" à esquerda e à direita do vão for selecionado.
m_iTipoPilarE m_iTipoPilarD	Atributos responsáveis por armazenar valores que irão definir o tipo de pilar selecionado pelo usuário. Eles assumirão o valor (0) ou (1) quando um dos Botões de Rádio "Morre" ou "Continua" for selecionado.

Tabela 6-2 Atributos da classe CArmacadAPP1

As funções membro próprias da classe CArmacadAPP1 são:

Nome da Função Membro	Descrição

afx_msg void OnPilarE()
afx_msg void OnPilarD()
afx_msg void OnVigaE()
afx_msg void OnVigaD()
afx_msg void OnBalancoE()
afx_msg void OnBalancoD()
afx_msg void OnPilarMorreE()
afx_msg void OnPilarContinuaE()
afx_msg void OnPilarMorreD();
afx_msg void OnPilarContinuaD();
	Funções responsáveis por enviar mensagens para a caixa de diálogos quando o usuário seleciona um dos Botões de Rádio existentes. Estas mensagens podem habilitar ou desabilitar os seus componentes, de forma a facilitar a interação com o usuário.

Tabela 6-3 - Métodos Próprios da classe CArmacadAPP1

Para que o usuário possa acessar as caixas de diálogos, foi necessário criar comandos específicos. Através do menu pull down, toolbar ou na linha de prompt, estes comandos serão acionados, chamando funções responsáveis por abrir as caixas de diálogos e proceder a passagem de dados. Foram então criadas as funções principais *void CriarDialogoVista()* e *void EditarDialogoVista()*.

Um exemplo de como os dados são transferidos das variáveis da classe CArmacadAPP1 para classe CVista, encontra-se definido no fragmento de código mostrado abaixo:

```
void CriarDialogoVista()
{
     // criação de um ponteiro para  a classe CVista
     CVista *pVista= new CVista;
     CAcModuleResourceOverride resOverride;
     // declaração de um objeto do tipo CArmacadAPP1
     CArmacadAPP1
dlg(CWnd::FromHandle(adsw_acadMainWnd()));
     if (dlg.DoModal()==IDOK)
     {
          // transferência dos valores fornecidos pelo
          usuário para
          //as variáveis da classe CVista, responsável
          pela criação //da entidade Vista.
          pVista->m_SetNomePeca(dlg.m_csNome);
          pVista->m_SetLarg(atof(dlg.m_csLargura));
          pVista->m_SetAlt(atof(dlg.m_csAltura));
```

```
            pVista->m_SetVao(atof(dlg.m_csVao));
            pVista-
>m_SetUnidades(atof(dlg.m_csQuantidade));

      ........
}
```

6.4 Interface da classe CCorte

Como já foi descrito no item 5.2.2, a classe CCorte é capaz de construir dois tipos de entidades distintas: a Viga em Corte e o Pilar em Corte. Portanto, deverão ser criadas duas entradas de dados diferentes. Cada uma delas será uma classe derivada da classe MFC CDialog: CArmacadAPP2 captura dos dados para construir a caixa de Viga em Corte e CArmacadAPP3, será a interface de criação da caixa do Pilar em Corte.

Cada uma delas possuirá atributos que terão um correspondente na classe CCorte ou na classe CCaixa.

Nome do atributo	**Descrição do que representa**
m_csNome m_csAltura m_csLargura m_csVao m_csQuantidade m_csApoioE m_csApoioD	Atributos responsáveis por armazenar os dados fornecidos pelo usuário através das caixas de edição encontradas no quadro de diálogos. Estes dados serão necessários para construir a representação gráfica da Viga em Corte.

Tabela 6-4 Atributos da classe CArmacadAPP2

Nome do atributo	**Descrição do que representa**
m_csNome m_csPeDireito m_csLargura m_csComprimento m_csQuantidade m_csLargura2	Atributos responsáveis por armazenar os dados fornecidos pelo usuário nas caixas de edição que aparecem neste quadro de diálogos. Estes dados serão necessários para construir a representação gráfica do Pilar em Corte.

m_csComprimento2 m_csDeltaX m_csDeltaY	

Tabela 6-5 Atributos da classe CArmacadAPP3

As classes CArmacadAPP2 e CArmacadAPP3 responsáveis pela interface da classe CCorte, não possuem botões de controle, portanto apenas as funções de sobrecarga obrigatória [**Tabela 6-1**] bastarão para construir estas entradas de dados.

As funções *voidCriarDialogoCorteViga()*, *voidEditarDialogoCorteViga()*, *voidCriarDialogoCortePilar()*, *voidEditarDialogoCortePilar()*, serão as responsáveis pela abertura dos quadros de diálogos e chamadas quando o usuário acionar os comandos para construir ou editar uma entidade do tipo CCorte.[**Figura 6-3]**e **[Figura 6-4**]

Figura 6-3 - Janela "Viga – Corte"

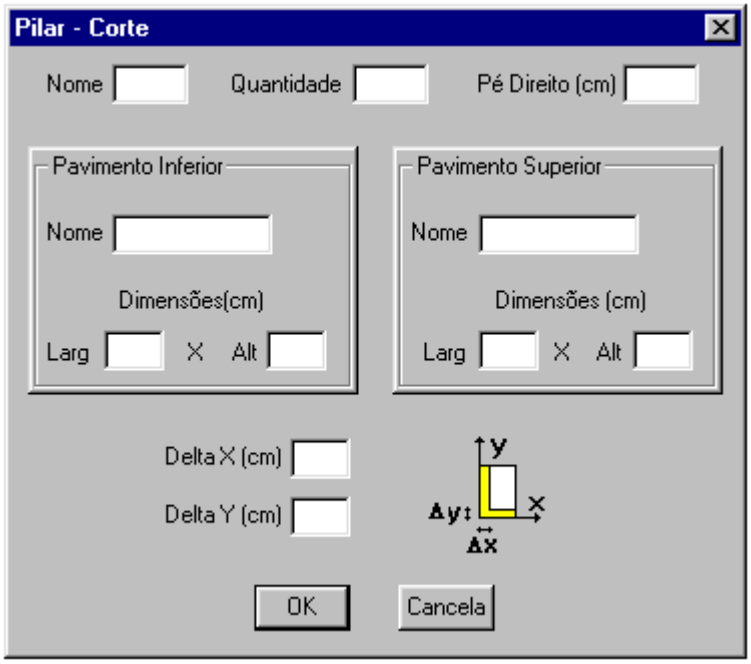


Figura 6-4 - Janela "Pilar – Corte"

6.5 Interface da classe CArmacaoVista

A classe CArmacadAPP5 será a interface da classe costumizada CArmacaoVista. Para desenhar a armadura de uma viga utilizando os mecanismos desta classe é necessário informar o cobrimento da ferragem, a área de ferro e o formato da barra da armadura.

Como já descrito no item 5.3.1, o aplicativo disporá de seis tipos de armaduras: Armadura Superior Principal, Armadura Superior Secundária, Armadura Inferior Principal, Armadura Inferior Secundária, Estribo, Costela. No quadro de diálogos, é fácil identificá-los, pois eles aparecem identificados com seus nomes.[**Figura 6-5**] O usuário deverá então informar a área de ferro calculada para cada tipo de armadura. Ao selecionar o botão de Rádio e escolher o formato da barra, logo aparece uma lista contendo opções de bitola e quantidade de barras correspondente a área informada. Para construir esta lista foram adotados alguns critérios, definidos conforme disposições construtivas e prescrições especiais de norma.

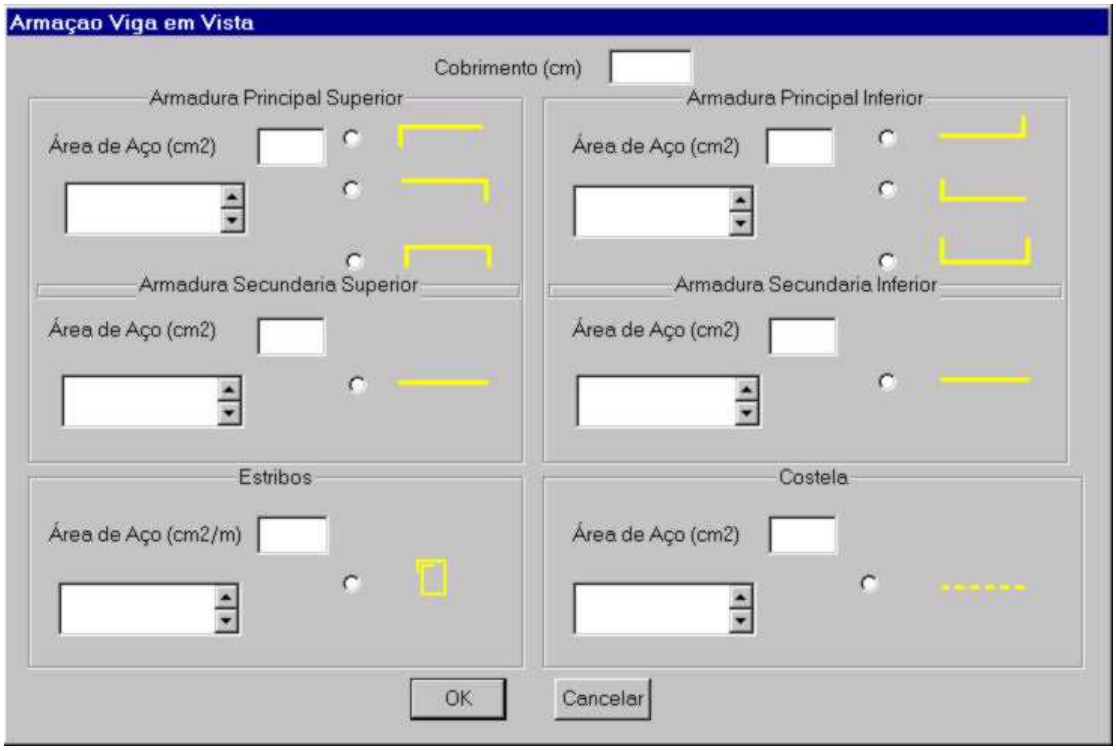


Figura 6-5 - Janela "Armação Viga em Vista"

Os critérios adotados para confeccionar a lista de seleção das armaduras longitudinais principais serão:

- O cálculo da quantidade de barras de cada bitola é feito dividindo-se a área total informada pelo usuário, pela área unitária de cada bitola definida pela norma EB-3. [**Tabela 6-6**] Conforme disposições construtivas, cada tipo de armadura deverá se constituir de pelo menos 2 barras.A quantidade de barras encontrada, através da divisão da área total pela área unitária, deverá ser sempre arredondada para cima, a favor da segurança.

∅(mm)	5	6.3	8	10	12.5	16	20	22.2	25	32
$A_s(cm^2)$	0.196	0.312	0.503	0.875	1.23	2.01	3.14	3.87	4.91	8.04

Tabela 6-6 - Área das barras empregadas como armaduras [Sus85]

- Para evitar o aparecimento de "ninhos" de concretagem, a NBR-6118 determinou o espaçamento mínimo entre as barras em uma mesma camada horizontal [**Figura 6-6**]:

  $e_h \geq$ 2cm (adotado)

  e

  $e_h \geq$ Diâmetro da barra longitudu isolada

  e

  $e_h \geq$ 1,2 do diâmetro do maior agregado utilizado para preparação do concreto (em torno de 1,5 cm) [Sus85]

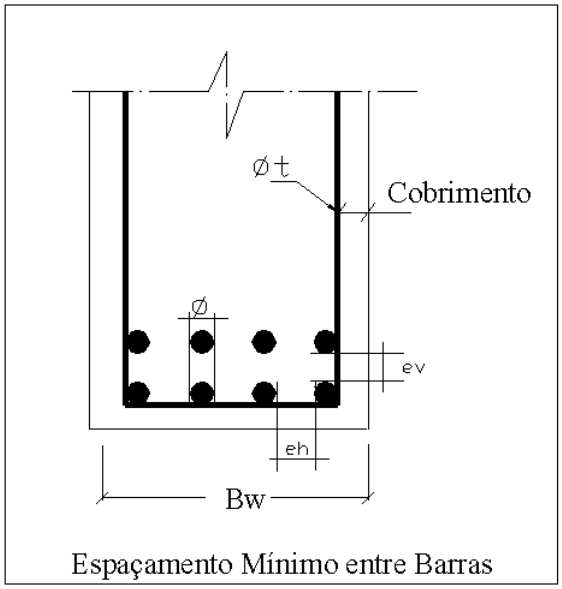


Figura 6-6

Neste aplicativo, o espaçamento mínimo adotado foi 2 cm pois, se forem usadas bitolas de até 20mm, é o que está mais a favor da segurança. A expressão abaixo fornecerá a largura necessária adotando-se para o diâmetro do estribo a bitola 8mm. Este valor será então comparado com a largura da viga a ser armada.

BMax = { [2 x (Cobrimento+0.8)]+[numBitola x ∅]+[2 x (numBitola-1)]}

Se m_dLarg ≥ BMax o espaçamento mínimo entre barras foi atendido.

- O comprimento de ancoragem também deverá ser testado. Por se tratar de barra inclinada a 90º com a horizontal, esta é uma situação de boa aderência. Deve-se calcular o comprimento de ancoragem com descrito a seguir:

  Cálculo da tensão de aderência τ_{bu} com η_b = 1.5 (coeficiente de aderência para o aço Ca50 e CA60)

  $$\tau_{bu} = 0.9 \sqrt[3]{(fcd)^2} \qquad fcd = fck/\lambda c$$

Cálculo do comprimento de ancoragem *lb* (boa aderência - sem gancho)

$$lb = (\varnothing/4)*(fyd/\tau_{bu}) \qquad fyd = fy/\lambda a$$

Este valor será comparado com o comprimento de ancoragem disponível.

Para o caso da armadura principal positiva:

$$lb_{dip} = m\_dLargApoio + [m\_dAlt - (2*Cobrimento)]$$

Para o caso da armadura principal negativa:

$$lb_{dip} = [m\_dAlt - (2*Cobrimento)]$$

Se $lb_{dip} \geq lb$ A ancoragem existente atende às especificações da NBR 6118.

Se estas exigências forem satisfeitas a string contendo o número e o diâmetro da bitola, será inserida na caixa de listagem e poderá ser selecionada.

No caso da Armadura Secundária Superior, as considerações são as seguintes:

- Cada tipo de armadura selecionada deverá se constituir de pelo menos 2 barras.
- Caso não tenha sido selecionada nenhuma armadura superior principal, esta armadura será apenas de composição. Neste caso os Botões de Rádio correspondentes a Armadura Principal Superior serão desativados.
- O espaçamento mínimo entre as barras deve ser calculado. Se existir armadura principal, testa-se primeiro se as duas podem compartilhar a mesma camada. Caso não seja possível, as barras secundárias ocuparão a segunda camada, e novamente o espaçamento mínimo será testado.
- O comprimento de ancoragem (***lb***) não será testado. O aplicativo não faz esta verificação cabendo ao usuário garantir se o valor da ancoragem é satisfatório.

Se forem atendidos os requisitos acima, a string contendo o número e o diâmetro da bitola, estará disponível para a seleção dentro da caixa de listagem.

No caso da Armadura Secundária Inferior, as condições impostas foram as seguintes:

- Cada tipo de armadura deverá se constituir de pelo menos 2 barras.
- A escolha de um tipo de Armadura Principal Inferior é obrigatória Se isto não ocorrer, ao tentar selecionar uma Armadura Secundária Inferior, o

usuário é avisado de que primeiramente, terá se selecionar a Armadura Principal.

- O teste do espaçamento mínimo, só poderá ser feito, quando a Armadura inferior Principal já estiver selecionada. Assim pode-se verificar, se a armadura secundária ficará na primeira ou na segunda camada.
- A exemplo da armadura Secundária Superior aqui o comprimento de ancoragem (***lb***) também não será testado.

A lista de seleção do estribo, a sua bitola e seu espaçamento deverão obedecer aos seguintes critérios:

- Só serão admitidos estribos com bitola menor ou igual a 8mm. Por questões construtivas o diâmetro máximo de um estribo deve obedecer às seguintes desigualdades. Isto pretende evitar que um estribo muito grosso, em viga fina , venha a prejudicar a concretagem.[Sus85]

  $\varnothing_t \leq b_w/12$ b_w = largura

  e

  $\varnothing_t \leq d/12$ d= altura - cobrimento

  Segundo as expressões sugeridas, uma viga fina com 10 cm de largura, só poderá comportar estribos com bitola abaixo de 8mm.
- Ao calcular o número de barras de uma determinada bitola, área fornecida pelo usuário deve ser dividida por 2, já que o estribo projetado possui duas pernas. O número de barras será calculado por metro, pois a área será fornecida em cm^2/m. Então, deve-se multiplicar o valor encontrado pelo comprimento do vão da viga fornecido em metros.

NumBitola = [(área de estribo / 2) /área unitária da bitola)] * vão

- Visando limitar e uniformizar ao longo da peça a abertura de fissuras oriundas do esforço cortante, a NBR 6118 recomenda um espaçamento máximo entre os estribos: [Sus85]

  $s \leq d/2$ d= altura- cobrimento

  e

  $s \leq 30$ cm

Para este aplicativo foi determinado que o espaçamento máximo entre os estribos será de15 cm. Tal espaçamento não atenderá a norma para vigas com d < 30 cm, mas devido a sua pequena dimensão não devem sofrer um esforço cortante muito grande, minimizando os efeitos descritos pela norma.

Foi necessário determinar um espaçamento mínimo, de forma a padronizar as saídas apresentadas na caixa de listagem Baseado na experiência o espaçamento mínimo adotado entre os estribos foi de 7,5 cm.

Depois de testadas, as opções de bitola e espaçamento de estribos poderão ser inseridas dentro da caixa de listagem para seleção.

Os critérios definidos para a apresentação das opções de número de barras e bitola para o Tipo de Armadura Costela são:

- Segundo NBR 6118 as vigas com altura útil (***d***) maior que 60 cm deve dispor longitudinalmente, nas faces laterais, de armadura de pele.[Sus85]. Tal armadura tem a função de "costurar" as fissuras que provavelmente surgirão devido à retração e variações térmicas. Portando, se a viga que está sendo armada apresentar altura menor que 60cm, o Botão de Rádio correspondente a este tipo de armadura estará desativado.
- Baseado na experiência concluiu-se que o diâmetro de 8 mm seria o máximo requerido para as costelas.
- Este tipo de armadura se apresenta aos pares, uma barra de cada lado ao longo da altura da viga. Portando a área fornecida pelo usuário em cm^2 deverá ser dividida por 2 para encontrar a quantidade de bitolas necessária.

Os atributos da classe CarmacadAPP5 serão:

Nome do atributo	**Descrição do que representa**
m_csCobrimento	Atributo que irá armazenar o valor do cobrimento da armadura.
m_csVao m_csAlt m_csLarg m_csApoioE m_csApoioD	Atributos que armazenam valores resgatados da classe CVista e CCaixa, necessários para construir as opções inseridas na caixa de listagem.
m_csAreaXXXX	Atributos responsáveis por armazenar a área fornecida pelo usuário nas caixas de edição disponíveis.
m_csItemKKKK	Atributos responsáveis por armazenar as strings selecionadas pelo usuário na caixa de seleção para cada tipo de armadura disponível.
m_csQuantBitolaYYYY	Atributos responsáveis por armazenar a quantidade de barras de uma mesma bitola selecionada pelo usuário na caixa de listagem.
m_csDiametroZZZZ	Atributos responsáveis por armazenar a diâmetro da barra selecionada pelo usuário na caixa de listagem.
m_ctListaFFFF	Atributos que representam cada uma das caixas de seleção apresentadas nesta Caixa de Diálogos

Tabela 6-7 Atributos da classe CArmacadAPP5

Os métodos Próprios da classe CArmacadAPP5 serão:

Nome da Função Membro	**Descrição**
CString GetItemXXXX()	Funções que retornam a string selecionada pelo usuário nas caixas de seleção para cada tipo de armadura.
afx_msg void OnArmaSup11() afx_msg void OnArmaSup12() afx_msg void OnArmaSup13() afx_msg void OnArmaSup21() afx_msg void OnArmaInf11() afx_msg void OnArmaInf12() afx_msg void OnArmaInf13() afx_msg void OnArmaInf21() afx_msg void OnArmaEstribo() afx_msg void OnArmaCostela()	Funções responsáveis por enviar mensagens para a caixa de diálogos quando o usuário seleciona um dos Botões de Rádio existentes. Estas mensagens podem habilitar ou desabilitar os seus componentes, de forma a facilitar a interação com o usuário

Tabela 6-8 - Métodos Próprios da classe CArmacadAPP5

As funções *voidCriarDialogoArmaVistaViga()* e *voidEditarDialogoArmaVistaViga(),* responsáveis pela abertura dos quadros de diálogos, serão chamadas quando o usuário acionar o comando para construir ou editar uma entidade do tipo CArmacaoVista.

6.6 Interface da classe CArmacaoCorte

A interface da classe CArmacaoCorte, será representada pela classe CArmacadAPP6 derivada da classe do MFC CDialog. O mecanismo de implementação desta classe segue os critérios e parâmetros já mencionados na classe CArmacadAPP5 que serve de interface para CArmacaoVista. Isto ocorre porque as duas representarão a entrada de dados para a construção da armação de vigas e ambas deverão seguir as mesmas condições.

Os atributos e as funções membros próprias da classe CArmacaoAPP6, é claro, serão semelhantes aos da classe CArmacaoAPP5 e, portanto, não vale a pena mencioná-los.

As funções *voidCriarDialogoArmaCorteViga()* e *voidEditarDialogoArmaCorteViga()*, serão chamadas quando o usuário acionar o comando para construir ou editar uma entidade do tipo CArmacaoCorte. [**Figura 6-7**]

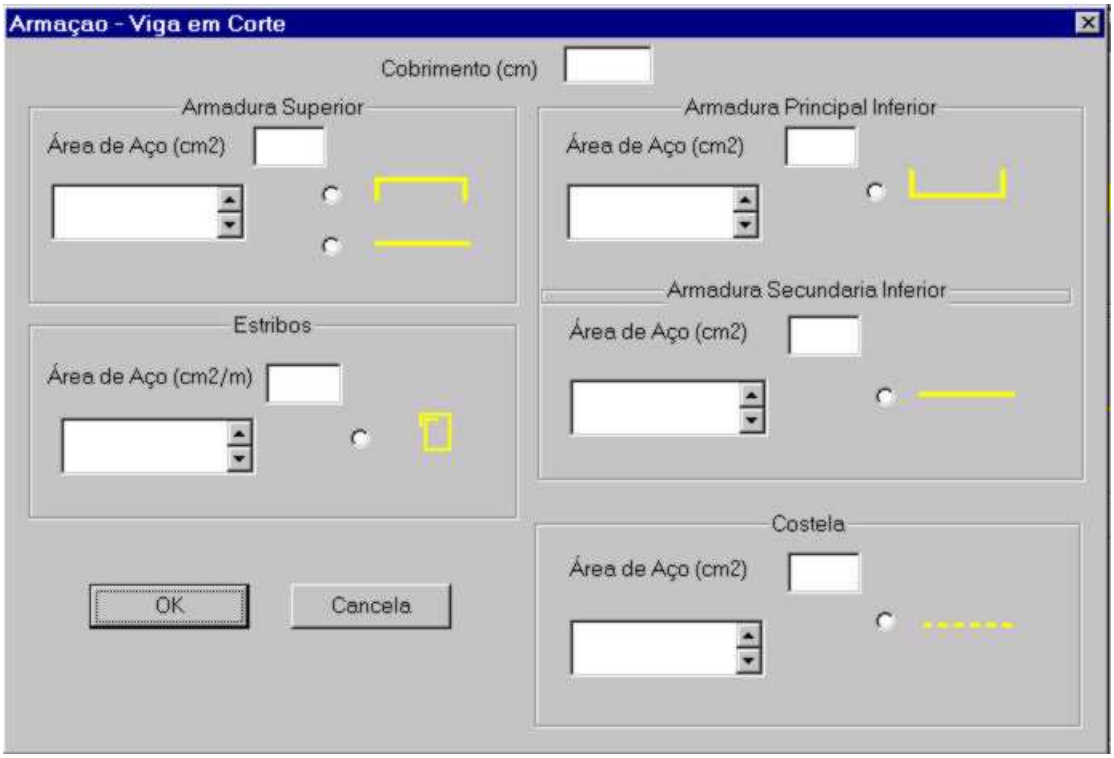


Figura 6-7 – Janela "Armação – Viga em Corte"

6.7 Interface da classe CArmacaoPilar

A interface da classe CArmacaoPilar, será representada pela classe CArmacadAPP7 derivada da classe do MFC *CDialog*.[Figura 6-8]

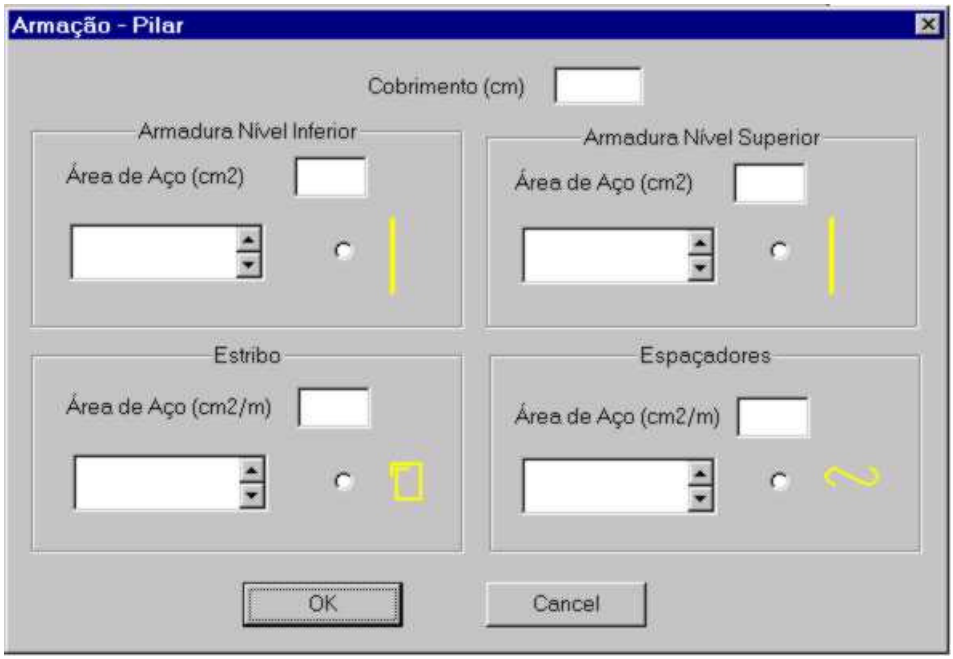


Figura 6-8 – Janela "Armação - Pilar"

Para desenhar a armadura de um pilar utilizando os mecanismos desta classe é necessário informar o cobrimento da ferragem e a área de ferro para os tipos de armadura disponíveis (Armadura Longitudinal do Pavimento Inferior, Armadura Longitudinal do Pavimento Superior, Estribos e Espaçadores) Ao selecionar o botão de Rádio, logo aparece uma lista contendo opções de bitola e quantidade de barras correspondente a área informada. Para construir esta lista foram adotados alguns critérios, definidos conforme disposições construtivas e prescrições especiais de norma.

Os critérios adotados para confeccionar a lista de seleção das armaduras longitudinais serão:

- Por questões construtivas a NBR6118 exige que a bitola da amadura longitudinal seja ao menos igual a 10mm. As barras da armadura não devem ser deformáveis quando estiverem no interior da forma, antes e durante a concretagem.[Süs91]
- O espaçamento máximo entre barras não deve ultrapassar 40 cm.
- O espaçamento horizontal mínimo admissível entre as barra é o mesmo adotado para as vigas [**Figura 6.6**]
- O espaçamento vertical mínimo admissível entre as barras também demonstrado na **Figura 6.6** obedecerá aos seguintes critérios:

$e_v \geq$ 2cm (adotado)

e

$e_v \geq$ Diâmetro da barra longitudinal isolada

e

$e_v \geq$ 0,5 do diâmetro do maior agregado utilizado para preparação do concreto (em torno de 1,5 cm) [Sus85]

Neste aplicativo, o espaçamento vertical e horizontal mínimo adotado foi 2 cm pois, se forem usadas bitolas até 20mm, este é o que está mais a favor da segurança.

- O comprimento de ancoragem será calculado como demonstrado no item 6.5, para a zona de boa aderência.

Os critérios adotados para confeccionar a lista de seleção da armadura do tipo Estribo serão:[Süs91]

- A bitola mínima para estribos de Pilar indicado pela NBR6118 é 5mm
- O espaçamento máximo entre os estribos não deverá superar os seguintes valores estabelecidos pela NBR6118: [Süs91]
  - 30 cm
  - menor dimensão externa da seção do pilar
  - $12\varnothing_L$ e $190\varnothing_T^2/\varnothing_L$ para os aços CA-40, CA-50 e CA-60. onde
    $\varnothing_L$ = diâmetro da armadura longitudinal.
    $\varnothing_T$ = diâmetro da armadura transversal.

  Neste aplicativo, a distância máxima adotada para o espaçamento dos estribos foi 20cm.

Os critérios adotados para confeccionar a lista de seleção da armadura do tipo Espaçadores serão:[Süs91]

- Só serão utilizados espaçadores em pilares cuja maior dimensão da seção transversal ultrapasse 60cm. Eles servirão para impedir a flambagem das barras comprimidas, funcionando como apoios intermediários.
- O mesmo espaçamento adotado para os estribos, deverá ser adotado para os espaçadores.

Os Atributos da Classe CarmacadAPP7 serão:

Nome do atributo	Descrição do que representa
m_csCobrimento	Atributo que irá armazenar o valor do cobrimento da armadura.
m_csPéDireito m_csLargPavInf m_csAltPavInf m_csLargPavSup m_csAltPavSup	Atributos que armazenam valores resgatados da classe CCorte necessários para construir a lista de seleção.
m_csAreaXXXX	Atributos responsáveis por armazenar a área fornecida pelo usuário nas caixas de edição disponíveis.
m_csItemKKKK	Atributos responsáveis por armazenar as strings selecionadas pelo usuário na caixa de seleção para cada tipo de armadura disponível.
m_csQuantBitolaYYYY	Atributos responsáveis por armazenar a quantidade de barras de uma mesma bitola selecionada pelo usuário na caixa de seleção.
m_csDiametroZZZZ	Atributos responsáveis por armazenar a diâmetro da barra selecionada pelo usuário na caixa de seleção
m_ctListaFFFF	Atributos que representam cada uma das caixas de seleção apresentadas nesta Caixa de Diálogos

Tabela 6-9 Atributos da classe CArmacadAPP7

Os Métodos Próprios da Classe CArmacadAPP7 serão:

Nome da Função Membro	**Descrição**
CString GetItemXXXX()	Funções que retornam a string selecionada pelo usuário nas caixas de seleção para cada tipo de armadura.
afx_msg void OnArmaPavInf() afx_msg void OnArmaPavSup() afx_msg void OnArmaEstribo() afx_msg void OnArmaEspaça()	Funções responsáveis por enviar mensagens para a caixa de diálogos quando o usuário seleciona um dos Botões de Rádio existentes. Depois que o usuário inserir a área da armadura,deverá acionar o botão de rádio correspondente para que a caixa de listagem seja preenchida seguindo os critérios já mencionados.

Tabela 6-10 - Métodos Próprios da classe CArmacadAPP7

As funções *voidCriarDialogoArmaPilar()* e *voidEditarDialogoArmaPilar(),* responsáveis pela abertura dos quadros de diálogos, serão chamadas quando o usuário acionar o comando para construir ou editar uma entidade do tipo CArmacaoPilar.

6.8 Interface da classe CLista

Não foi preciso criar uma caixa de diálogos para construir a interface da classe CLista. Como já descrito no item 5.4, a lista de materiais será construída ao mesmo tempo em que as armaduras das peças estão sendo criadas. Depois que o desenho já estiver completo, basta chamar as funções *voidCriarLista()* e *voidCriarResumo()* acionando os comandos na linha de prompt do AutoCAD , no Toolbar ou no menu "pool down" correspondente.

Apêndice I

INTEGRAÇÃO ENTRE O AMBIENTE DE PROGRAMAÇÃO VISUAL C++ E AS BIBLIOTECAS OBJECTARX E MFC

A Construção de um Projeto no Visual C++ 6.0 integrando a Biblioteca ObjectARX 2000

A Construção de um Projeto no Visual C++ 6.0 integrando a Biblioteca MFC (Microsoft Fundation Class) e a Biblioteca ObjectARX 2000

4.1 A Construção de um Projeto no Visual C++ 6.0 integrando a Biblioteca ObjectARX 2000

(Configuração de um projeto no ambiente de programação Visual C++ 6.0 para geração de uma aplicação .arx)

Uma aplicação em ObjectARX é uma DLL (*Dynamic Linked Library*) com extensão *.arx*. Este arquivo é composto basicamente de dados e funções, compilados, que se tornam parte do AutoCAD. Um Workspace é um ambiente de trabalho criado no Visual C++ capaz de armazenar projetos.

Para que o Visual C++ possa produzir este tipo de arquivo executável, é necessário configurá-lo de forma adequada. A Autodesk fornece parâmetros para executar esta operação:

a) Criar um novo projeto:

Abra o programa Visual C++ 6.0 e no menu “pull down” "File" selecione "New". Na caixa de diálogos "New"; selecione a tabela "Projects".

Selecione o tipo "Win32 Dynamic-Link Library" na tabela de tipos de projetos disponíveis.

Coloque o nome do seu projeto no campo "Project Name", verifique se o diretório "Location" é o desejado para salvar o projeto e clique "OK".

Na nova caixa de diálogos "Step 1 of 1"escolha "An empty DLL project". Clique em "Finish" e depois em "OK".

b) Fornecer os dados para compilação do projeto:

No menu “pull down” "Project", selecione "Settings" para abrir a caixa de diálogos "Project Settings".

No campo "Settings For" selecione a opção "All Configurations" para que a mesma configuração possa ser feita para as versões "Debug" e "Release" do aplicativo.

Certifique se o nome do seu projeto aparece selecionado na árvore mostrada abaixo da caixa "Settings For".

Verifique se ao ativar a etiqueta "General", a opção selecionada na caixa "Microsoft Foundation Classes" é a "Not Using MFC". Neste estágio, o Visual C++ está sendo configurando sem a Biblioteca MFC.

Selecione a etiqueta "C/C++" e no campo "Category" selecione "Code Generation". No campo "Use run_time library" selecione a opção "Multihreaded DLL".

Ainda na etiqueta "C/C++", selecione no campo "Category" a opção "Preprocessador". No campo "Additional include directories" adicione o caminho dos arquivos "include" do ObjectARX.
Exemplo: C:\ObjectARX 2000\inc.

c) Configurar os dados de linkagem do projeto:

No menu “pull down” "Project", selecione "Settings" para abrir a caixa de diálogos "Project Settings".

Selecione a etiqueta "Link".

Certifique-se que no campo "Settings For" está selecionada a opção "All Configurations". Esta opção deve estar sempre selecionada.

Selecione no campo "Category" a opção "General" e preencha o campo "Output file name" com o nome do aplicativo *arx* que você deseja gerar. Exemplo: Armacad.arx.

O campo "Object/library modules" deverá abrigar o nome dos arquivos .lib existentes no subdiretório ObjectARX 2000\lib. Estas *libraries* são requeridas durante a linkagem da aplicação dependendo da classe do ObjectARX que está sendo usada. Existem alguns destes arquivos que deverão ser obrigatoriamente mencionados: rxapi.lib, acrx15.lib, acutil15.lib, e o acedapi.lib. Os outros poderão também ser inseridos bastando copiar seus nomes do diretório .lib do ObjectARX.

Outra forma de adicionar os arquivos .lib no projeto é acessar o menu pull down "Tools" a opção "Options". Na etiqueta "Directories" acrescente o caminho do diretório onde estão os arquivos .lib do ObjectARX 2000. Exemplo: C:\ObjectARX 2000\lib. Neste caso, não é necessário acrescentar

estes arquivos na caixa "Object/library modules" e todas as *librarys* estarão disponíveis para o aplicativo.

Selecione agora, no campo "Category" a opção "Input". No campo "Additional Library Path" coloque também o caminho do diretório onde estão os arquivos .lib do ObjectARX.

Feche a caixa de diálogos "Project Settings" clicando no botão "OK".

d) Criar e inserir no projeto um conjunto mínimo de pastas:

Na tela principal do Visual C++ existem duas maneiras de inserir pastas em projeto. Através do menu "Project", submenu "Add to Project", "New Folder", ou clicando o botão direito do mouse , apontado para a pasta do projeto, acionando o submenu "New Folder". Nas duas alternativas pode-se visualizar a janela "New Folder" que contem os campos "Name of the new folder" - fornecer o nome da pasta - e "File extensions" - fornecer a extensão dos arquivos que irão compor esta pasta.

Criar quatro pastas com as seguintes sugestões de nomes e as extensões dos arquivos componentes:

- Pasta: “Source”; Extensão: .cpp - Seus arquivos abrigarão a implementação das classes que irão compor o projeto. Esta pasta ainda guardará um arquivo com extensão *.def*.
- Pasta: “Header”; Extensão: .h - Neste caso, os arquivos aqui armazenados serão responsáveis pela declaração das classes e funções do projeto. Quando o programa está sendo compilado, são estes arquivos os responsáveis pela troca de informações entre os diversos módulos do programa.
- Pasta: “Library”; Extensão: .lib - Inserir os arquivos contidos no subdiretório *lib* do ObjectARX exceto o arquivo RXHEAP.LIB.
- Pasta: “ARX”; Extensão: .h - Inserir os arquivos contidos no subdiretório *inc* do ObjectARX selecionando apenas os de terminação *.h*. Isto permitirá o uso de um recurso no Visual C++ que permite visualizar as funções e atributos de um objeto de uma classe, durante a sua implementação.[**Figura I-1**]

e) **Criar e adicionar os arquivos indispensáveis para o funcionamento do aplicativo:**

Para criar um arquivo com a terminação .cpp , no menu “pull down” "Project" selecione "Add to Project" e então "New".

Depois que a janela "New" estiver aberta, selecione o tipo de arquivo desejado "C++ Source File" e preencha o quadro "File name" com o nome do arquivo.

Este arquivo poderá ser incluído no projeto bastando clicar com o botão direito do mouse na Pasta "Source", na opção "Add Files to Folder".

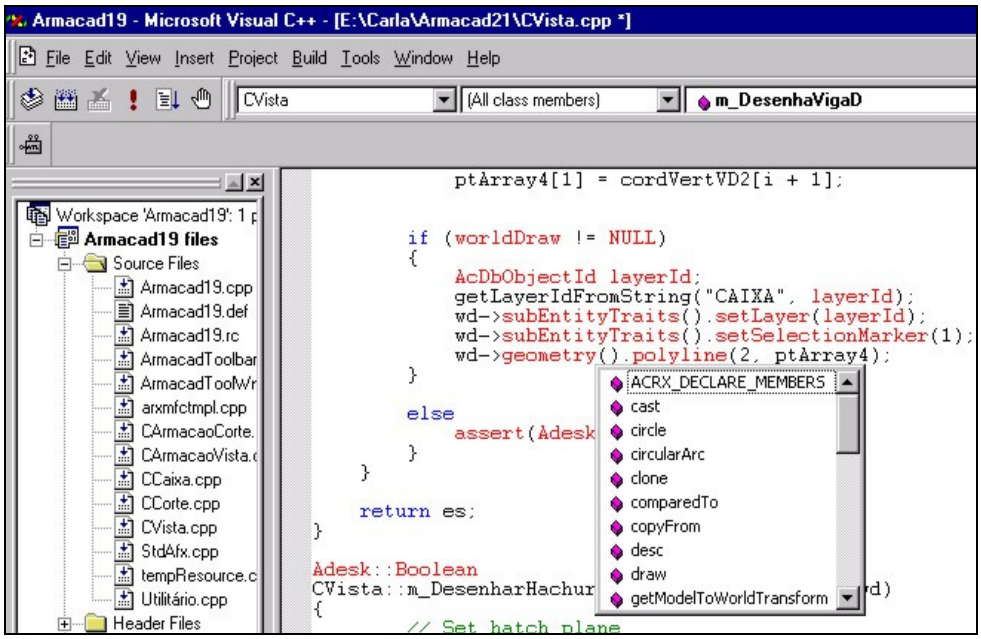


Figura I-1 –Visualização das funções e atributos

Código mínimo do arquivo *.cpp*: Ele deverá conter a declaração e a implementação das funções *initApp ()* e *unloadApp()*. Elas são responsáveis pelo registro e remoção dos comandos e classes costumizadas usadas no aplicativo.

Ele também deverá conter a função global *acrxEntryPoint*. O AutoCAD chama esta função para receber as mensagens da aplicação.

Um exemplo de código para estas funções pode ser encontrado no Guia do Desenvolvedor [Aut99].

Criação do arquivo *.def* : Ele é responsável pela exportação das funções acrxEntryPoint() e acxrGetApiVersion() para que o AutoCAD possa acessá-las. Selecione o menu pull down "Project", a opção "Add to Project" e "New" na caixa de diálogos. O tipo de arquivo a escolher é "Text File". Na caixa "File

name" coloque o nome usado para denominar a sua aplicação *.arx*. Exemplo: Armacad. É necessário o nome seja o mesmo dado para a aplicação quando da configuração do projeto. Salvar este arquivo com a terminação *.def*. Ele deverá ter o seguinte formato:

```
DESCRIPTION     ( descrição do aplicativo - nome, versão)
LIBRARY         <projectName> (nome dado à aplicação .arx)
EXPORTS

                acrxEntryPoint          PRIVATE
                acrxGetApiVersion       PRIVATE
```

Outra dica importante, pois ajuda a evitar erros de digitação: O Visual C++ possui uma forma de destacar as palavras chave do código utilizando a cor da letra que aparece na janela de edição. Cada cor poderá corresponder a um tipo de variável. Por exemplo, azul pode ser a cor das palavras reservadas da linguagem (if,for). Para que o Visual C++ possa reconhecer as palavras exclusivas da bilbioteca ObjectARX, é necessário verificar a existência do arquivo *usertype.dat* no diretório \VisualStudio\Common\MSDev98\Bin. Em caso afirmativo, basta escolher a cor desejada, selecionando no menu “pull down” "Tools", submenu "Options", tabela "Format", escolhendo no campo "Colors", a cor desejada para as variável do tipo "User Defined Keywords". Assim, se a palavra reservada não estiver escrita corretamente, ela não assumirá a cor escolhida.[**Figura 4-2**].

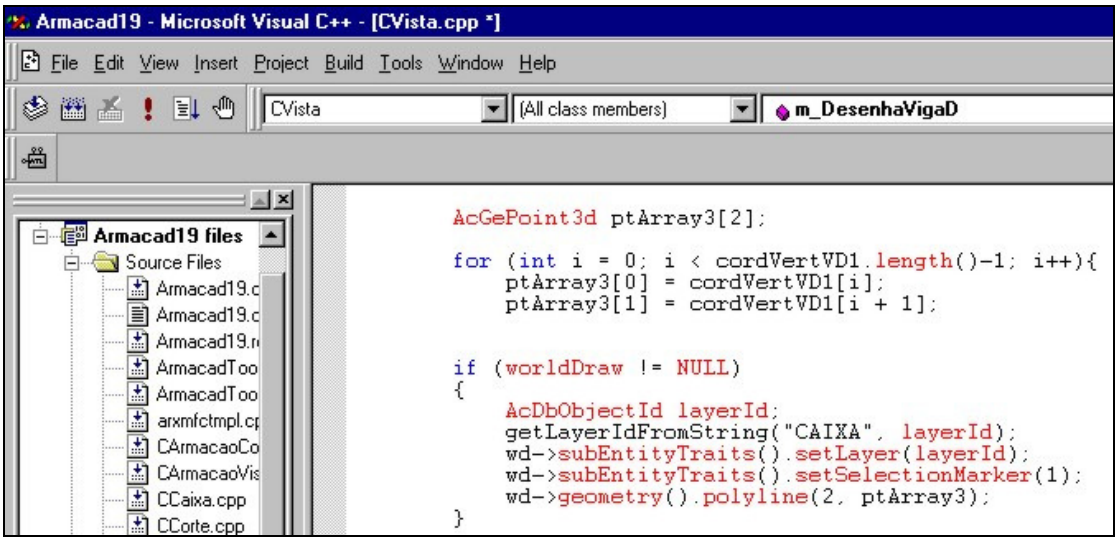


Figura I-2 – Visualização das cores nas palavras reservadas

A Autodesk desenvolveu um aplicativo de ajuda (*wizard*) para criar e desenvolver projetos usando o ObjectARX. Neste caso, a configuração do Visual C++ sofrerá algumas alterações. No início do desenvolvimento deste trabalho, a versão apresentava alguns problemas. Optou-se por não usar tal aplicativo.

4.2 A Construção de um Projeto no Visual C++ 6.0 integrando a Biblioteca MFC (Microsoft Fundation Class) e a Biblioteca ObjectARX 2000.

(Configuração de um projeto no ambiente de programação Visual C++ 6.0 para geração de uma aplicação .arx para utilizar a Biblioteca MFC)

Uma aplicação usando o a biblioteca ObjectARX pode ser criada possibilitando a utilização da biblioteca MFC (Microsoft Foundation Class). Neste caso, para construir um projeto visando uma aplicação que use como interface, janelas no padrão do sistema Windows, deve-se seguir alguns passos:

a) Criar um novo projeto:

Abra o programa Visual C++ 6.0 e no menu “pull down” "File" selecione "New". Na caixa de diálogos "New"; selecione a tabela "Projects".

Selecione o tipo "MFC AppWizard (DLL)" na tabela de tipos de projetos disponíveis.

Coloque o nome do seu projeto no campo "Project Name", verifique se o diretório "Location" é o desejado para salvar o projeto e clique "OK".

Na nova caixa de diálogos "Step 1 of 1” escolha "MFC Extension DLL (using shared MFC DLL)". Clique em "Finish" e depois em "OK".

Os dados de compilação e linkagem do projeto são os mesmos utilizados no item 4.1, portanto não serão repetidos aqui.

As pastas a serem inseridas, com os respectivos arquivos, seguirão a orientação adotada no item 4-1.

b) Criar e adicionar os arquivos indispensáveis para o funcionamento do aplicativo:

Com a configuração do projeto, aparecem automaticamente alguns arquivos inseridos no projeto. Dentro deles, pode-se encontrar códigos que deverão ser modificados para o perfeito funcionamento do aplicativo.

Arquivo de terminação *.cpp*

Este arquivo, que geralmente possui o mesmo nome dado ao projeto, será o arquivo principal do projeto. Como definido no item 4-1, as funções *inipAPP*, *unloadAP* e terão as suas implementações inseridas neste arquivo.

Além dos já existentes, novos arquivos de tipo .h, deverão ser incluídos através da palavra reservada da linguagem "include". São eles: o *AcExtensionModule.h* e o arquivo de declaração das classes derivadas do MFC, responsáveis pela interface do aplicativo. Ex: ArmacadAPP.h.

A função *extern "C" HWND adsw_acadMainWnd ()* deverá ser declarada logo após as definições de compilação como demonstrado abaixo:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
extern "C" HWND adsw_acadMainWnd();
```

A função *DLLMain()* encontrada neste arquivo, deverá ter o seu código alterado para a forma descrita abaixo:

```
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID
lpReserved)
{
    UNREFERENCED_PARAMETER(lpReserved);
    if (dwReason == DLL_PROCESS_ATTACH)
    {
     (Nome do AplicativoDLL).AttachInstance
     (hInstance);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
     (Nome do AplicativoDLL).DetachInstance();
```

```
    }
    return 1
}
```

A função *acrxEntryPoint()* também deverá ser inserida com o seguinte formato:

```
extern "C" AcRx::AppRetCode acrxEntryPoint
( AcRx::AppMsgCode msg, void* appId)
{
    switch( msg )
    {
    case AcRx::kInitAppMsg:
     acrxDynamicLinker->unlockApplication(appId);
     acrxDynamicLinker->registerAppMDIAware(appId);
        initApp();
        break;
    case AcRx::kUnloadAppMsg:
        unloadApp();
        break;
    case AcRx::kInitDialogMsg:
        break;
    default:
        break;
    }
    return AcRx::kRetOK;
}
```

O arquivo *.def* terá as mesmas características descritas no item.4-1

No arquivo *StdAfx.h* que faz parte do novo projeto criado, deve-se incluir as diretivas de compilação, no início do arquivo:

```
#ifdef _DEBUG
#define WAS_DEBUG
#undef _DEBUG
#endif
```

e no final do arquivo:

```
#ifdef WAS_DEBUG
#define _DEBUG
#undef WAS_DEBUG
#endif
```

O aplicativo de ajuda (*wizard*) para criar e desenvolver projetos usando o ObjectARX, é capaz de configurar automaticamente o Visual C++ habilitando-o para utilizar a biblioteca MFC. Neste projeto, como já foi dito, optou-se por não usar tal aplicativo.