UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

UMA REPRESENTAÇÃO DE DADOS UTILIZANDO O PARADIGMA DE ORIENTAÇÃO A OBJETOS

POR : MARCÉLIA CARVALHO REZENDE

ORIENTADOR : MÁRCIO LUIZ DE ANDRADE NETTO

Tese apresentada à Faculdade de Engenharia Elétrica -FEE-UNICAMP, como parte dos requisitos exigidos para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA

- Dezembro de 1990 -

" Uma longa jornada começa
com o primeiro passo "

dedico este trabalho

aos meus pais
José Carlos e Jurany
aos meus irmãos
Lucita e José
ao Viktor

AGRADECIMENTOS

- Ao professor Márcio Luiz de Andrade Netto, pela orientação ao longo do desenvolvimento deste trabalho,

- Ao CNPq e FAEP, pelo apoio financeiro,

- Aos amigos Roberto Hiroshi, Sérgio Braga, Hsieh e José Luiz, cujo convívio e amizade estarão gravados por toda minha vida,

- Aos colegas que ajudaram a fazer esta jornada mais gratificante, em especial a Humberto e Verônica,

- Aos meus pais pela constante confiança e dedicação em mim depositadas,

- Ao Viktor, pelo seu apoio, respeito, confiança e dedicação, demonstrados ao longo de nosso convívio,

- A todos que direta e indiretamente colaboraram para execução deste trabalho.

RESUMO

Este trabalho apresenta a metodologia de orientação a objeto para desenvolvimento de sistemas de software.

Discute-se a implementação de um sistema desenvolvido a partir da combinação de objetos que são instâncias de classes definidas utilizando-se técnicas de orientação a objeto.

Tal sistema tem suporte para representar dados complexos como estruturas de "frames". Estas estruturas de "frames" são representadas a partir da combinação de tipos previamente definidos.

ÍNDICE

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

1.1. METODOLOGIA DE ORIENTAÇÃO A OBJETOS

Um dos problemas críticos no desenvolvimento de um software reside na manutenção, onde o custo chega a 70% do custo total. Isto indica uma ineficiência na metodologia de projeto e implementação de software.

A manutenção de um sistema consiste, principalmente, na alteração do mesmo em decorrência de novas especificações e/ou modificações no formato de dados. Esta manutenção, geralmente, acarreta grandes modificações ao longo de todo o sistema, indicando que metodologias adotadas para o desenvolvimento de sistemas tornam-os altamente centralizados e que a implementação dos dados é distribuída ao longo de todo um sistema. O ideal seria que a manutenção consistisse numa alteração rápida e realizada apenas em partes do sistema.

Ao desenvolver um novo sistema, o que frequentemente ocorre, é iniciar o projeto e implementação da estaca zero, apesar de muitas vezes haver tarefas semelhantes, e já desenvolvidas para utilização em outros sistemas.

Com os aspectos considerados anteriormente, o que se vê é uma inadequação dos métodos utilizados para um desenvolvimento de sistemas de software significantes.

Resolver o problema de extensão e reutilização é uma tarefa significativa para uma evolução na área de software.

A Metodologia de Orientação a Objeto, adotada para o desenvolvimento de projetos de software, considera como ponto central os dados que os mesmos manipulam e é caracterizada por certas técnicas. Estas técnicas, quando utilizadas adequadamente, permitem criar sistemas a partir da combinação de módulos. Estes módulos são descentralizados, ou seja, possuem um papel bem definido dentro de um sistema, contendo dados e procedimentos disponíveis, favorecendo uma expansão do sistema e uma reutilização destes módulos em novos sistemas.

Tal metodologia faz uso de um mecanismo de herança que permite estabelecer relacionamentos entre módulos e, juntamente com o polimorfismo, estabelece um maior nível de abstração deste relacionamento.

1.2. REPRESENTAÇÃO DE INFORMAÇÕES

A representação de informações constitui uma tarefa de constante pesquisa, no meio da ciência da computação [AHO83] [ULL82] [KOR86]. Tal empenho tem a finalidade de modelar informações de problemas reais em programas de aplicação em computador de modo a satisfazer certas exigências, tais como :

1- fidelidade na representação : que o modelo contenha todas as características da informação;

2- facilidade no manuseio;
3- eficiência na execução.

Na aplicação em banco de dados, há modelos de dados já bem formalizados, como : Relacional, Hierárquico, Network e Entidade_Relacionamento [ULL82] [KOR86]. Tais modelos foram introduzidos visando atender às exigências de sistemas de banco de dados comerciais.

Em Aplicações como Inteligência Artificial (IA), Projeto Auxiliado por Computador (PAC) e ambientes orientados a objetos, há exigências de modelos para representar informações que não tenham a mesma homogeneidade estrutural que a apresentada em aplicações comerciais. Logo, vê-se a necessidade de novos modelos.

Propostas de modelos orientados por objetos tentam satisfazer exigências de certas aplicações [XAV90] [FON90] [BAR90]. Estes modelos trazem consigo características como encapsulamento de informações e procedimentos, abstrações como generalização e especialização e, decisão, em tempo de execução, de qual tarefa ativar. Estas características permitem :

1- a representação de dados complexos de forma que os mesmos possam ser tratados como objetos;

2- a representação de dados dinâmicos (dados que têm seu esquema em constante evolução, junto com a própria aplicação);

3- que as aplicações preocupem-se apenas com *quais* ações devem ser executadas por um objeto, sem considerar *como* estas ações devem ser executadas.

Porém, tais modelos não possuem ainda nenhum formalismo básico e não estão determinadas todas as características que os mesmos devem ter [BAN88].

1.3. OBJETIVOS DO TRABALHO

A metodologia de Orientação a Objetos bem como as linguagens de programação que suportam a orientação a objetos são mecanismos adequados para projetar e implementar modelos de dados como objetos, pois as características que estes modelos devem satisfazer podem ser definidas e implementadas de maneira natural e eficiente, já que a linguagem orientada por objeto deve suportar definição de tipos abstratos, abstrações (como generalização/especialização) e polimorfismo.

Assim sendo, o trabalho desenvolvido tem como objetivo aplicar a metodologia de orientação a objetos para desenvolver um sistema que cria tipos a partir de 10 tipos pré_definidos. Estes tipos têm suporte para representar dados relativamente complexos, sendo possível estruturar representações de dados como "frames".

Depois que os dados estão representados, é possível manipulá-los através da apresentação na tela do tipo definido. A partir daí, dados correspondentes a esta representação podem ser lidos, guardados e removidos de arquivos.

O sistema é implementado utilizando-se uma linguagem que tem suporte para aplicar as técnicas de orientação a objetos (no caso C++).

Alguns módulos são definidos, e posteriormente combinados para constituir o sistema proposto. Estes módulos tentam satisfazer certas características para que possam ser reutilizados em outros sistemas diferentes a este sistema onde eles foram inicialmente definidos.

A dissertação é estruturada da seguinte forma :

▸ Capítulo 1 - o presente capítulo.

▸ Capítulo 2 - apresenta aspectos de qualidade de software e descreve as características de uma linguagem orientada por objeto, especificando as quatro técnicas que a Metodologia de Orientação a Objeto suporta : Encapsulamento, Abstração de Dados, Mecanismo de Herança e Polimorfismo.
O conceito de TIPO ABSTRATO DE DADO é apresentado e é introduzido os termos CLASSE e OBJETO.

▸ Capítulo 3 - apresenta o trabalho, descrevendo os módulos implementados, que combinados, adequadamente, constituem o sistema proposto.
São definidos e implementados 10 tipos para que possam ser combinados para representar dados relativamente complexos. Os tipos são os seguintes : String, Inteiro, Double, Longo, Vazio, Produto Vetorial, Lista, Btree, Menu e União.
Os quatro primeiros tipos (String, Inteiro, Double e Longo) já estão definidos na linguagem C++, mas foram reconsiderados para uma melhor uniformidade no tratamento dos tipos e também pelo fato que tais tipos são considerados juntamente com procedimentos e características dotando-os de semântica.
Uma vez especificado um tipo para representar um certa informação, através da classe CAMPO_MENS é possível manipular com a estrutura que contém todas as informações que definem tal tipo. Esta estrutura é denominada **campo_mensageiro**.
Através da classe CAMPO é possível declarar objetos que manipulam com dados que são representados por um tipo definido previamente. Tal classe foi implementada explorando intensamente a técnica de polimorfismo, para que diferentes tipos de dados possam ter diferentes procedimentos utilizando-se o mesmo nome de funções.
E finalmente, através das classes INTERF, MENU e EDICÃO é possível apresentar uma interface adequada à informação representada, para que usuários tenham acesso aos dados armazenados em arquivo. Tais dados são armazenados utilizando-se objetos da classe FICHÁRIO E REMENDO, que são duas classes responsáveis pela manipulação de dados em disco.

▸ Capítulo 4 - descreve exemplos de aplicação utilizando os módulos desenvolvidos.

▸ Capítulo 5 - enumera as qualidades e deficiências do trabalho, as vantagens alcançadas ao desenvolver tal trabalho e sugestões de trabalhos futuros.

CAPÍTULO 2

ASPECTOS DE QUALIDADE DE SOFTWARE E PROGRAMAÇÃO ORIENTADA POR OBJETO

2.1. GENERALIDADES

Um sistema de software tem que satisfazer qualidades para atender as exigências tanto do usuário do sistema quanto do especialista que projeta o sistema. São qualidades como facilidade para utilizar o sistema, sua rapidez de execução, facilidade de extensão, de acordo com novas especificações do produto e uma reutilização das partes componentes (ou módulos) em novos sistemas.

A preocupação com estes aspectos leva não só à formalização de metodologias que procuram facilitar o alcance dos mesmos, como também, ao desenvolvimento de linguagens com ferramentas adequadas para suportar a aplicação destas metodologias [MEY88].

Este capítulo descreve alguns aspectos de qualidade de software e as facilidades que a metodologia de orientação a objeto oferece para atingi-los, descreve as características de uma linguagem que suporta tal metodologia enumerando algumas destas linguagens e finalmente apresenta, de maneira geral, o trabalho desenvolvido.

2.2. ASPECTOS DE QUALIDADE DE SOFTWARE

Quando se fala em sistemas de software, espera-se sempre que estes possuam certas qualidades, tais como : rapidez de execução, confiabilidade, facilidade de uso, legibilidade e modularidade.

Estas qualidades podem ser separadas em dois grupos :

1 - qualidades externas : detectadas pelo usuário do produto.
ex : rapidez de execução e facilidade de uso;

2 - qualidades internas : detectadas pelo especialista em com putação.
ex : legibilidade e modularidade.

Na realidade, são os fatores externos que interessam no final. Porém, é através de projetos bem feitos que se consegue, a longo prazo, manter as qualidades externas de um produto de software, ou seja, são as qualidades internas que fazem com que as qualidades externas sejam realmente atingidas.

Também, levando-se em consideração que 70% do custo de um software está na manutenção [MEY88], principalmente no que se refere à atualização do sistema decorrente de modificações nas especificações e de modificação no formato dos dados, realmente deve-se desenvolver um método de projeto que diminua tal custo.

Fazendo-se uma análise destes custos, percebe-se como os sistemas são centralizados, já que uma alteração na especificação leva a uma sequência de alterações ao longo do produto de software. Outro fator que se percebe éque programas têm acesso aos dados através de sua implementação ao invés de acesso através de suas propriedades, pois

quando há modificação no formato dos dados, há uma modificação nos programas que fazem uso destes dados.

Tornar possível a extensão de sistemas sem muito custo e fazer com que estes sejam formados pela composição de módulos que podem ser reutilizados em outro sistema é uma tarefa significativa para uma evolução na área de software.

O aspecto de reutilização de um software é um tema muito discutido já há bastante tempo. Em 1968, D. McIlroy [McI76] abordou a produção em série de componentes de software. Ainda hoje é um sonho imaginar em se ter catálogos de módulos de software da mesma forma que há catálogos de circuitos VLSI, porém há tentativas para tornar este sonho uma realidade. As tentativas iniciais não resolveram o problema, devido a sua aplicabilidade bastante limitada, entretanto serviram para mostrar os aspectos importantes do problema da reutilização :

1 - Reutilização de códigos fontes - é uma ocorrência comum em meios acadêmicos. Permite a avaliação de códigos de programas sendo possível estendê-los, estudá-los ou imitá-los;

2 - Reutilização de pessoal - é uma ocorrência comum na indústria. Há uma transferência de engenheiros de software de um projeto para outro, com o objetivo de auxiliar um novo projeto com as experiências adquiridas em projetos anteriores;

3 - Reutilização de projeto - outra ocorrência comum na indústria. Projetos de aplicações mais comuns, que já foram desenvolvidos, são guardados para auxiliar em projetos futuros.

Observando as considerações feitas, busca-se desenvolver uma metodologia, que uma vez aplicada na fase de projeto e implementação de um sistema, este tenha qualidades, tais como : extensão e reutilização.

A metodologia de orientação a objetos vem atender a tais anseios. Ela baseia-se na criação de módulos utilizando o conceito de TIPO ABSTRATO DE DADO e na aplicação de técnicas de polimorfismo e mecanismo de herança.

Os módulos tornam os sistemas descentralizados, pois cada módulo contém dados com suas características, propriedades e serviços disponíveis. Desta forma, fica claramente estabelecido quais módulos são responsáveis por quais tarefas dentro de um sistema. As técnicas de polimorfismo e mecanismo de herança permitem a reutilização de módulos com um certo grau de abstração a nível de programação. É o uso das técnicas em conjunto que favorecem a extensão de um sistema.

Finalmente, cabe ressaltar que um projeto orientado a objetos, para ter um bom desempenho, deve ser implementado numa linguagem que ofereça um suporte apropriado para se definir os módulos e aplicar as técnicas de polimorfismo e herança. Um comportamento onde é feito um projeto orientado a objeto e sua implementação numa linguagem como C ou Pascal, não há uma adequação em desenvolvimentos significantes. Com o tempo, a medida que ocorre a evolução e manutenção de um sistema, a distância ("gap") entre as idéias do seu projeto e sua realização vai se tornando maior.

2.3. Programação Orientada por Objeto

Segundo Stroustrup [STR88], uma linguagem suporta uma certa técnica de programação quando possui facilidades intrínsecas para que esta técnica possa ser implementada sem muito esforço e de forma segura e eficiente.

Em particular, a metodologia de orientação a objetos abrange certas técnicas que exigem certos recursos de uma linguagem para que ela possa ser considerada ORIENTADA A OBJETOS.

A metodologia é caracterizada por 4 técnicas : encapsulamento, abstração de dados, mecanismo de herança e polimorfismo. Através destas técnicas torna-se possível a reutilização das partes de um sistema. Com o mecanismo de herança e polimorfismo a extensão de um sistema é um processo bastante natural, sendo que o polimorfismo permite uma considerável abstração a nível de programação.

2.3.1. ENCAPSULAMENTO E DADOS ESCONDIDOS

Termos em inglês : Data hiding, Encapsulation.

Um sistema de software é um mecanismo de execução de certas ações sobre certos dados.

O projetista de um sistema depara sempre com a questão : "O sistema deve ser baseado em dados ou em ações?". Ao responder a esta pergunta ele está optando por uma metodologia para projetar seu sistema.

O método tradicionalmente usado é o método baseado em ações, onde estas têm uma importância principal e, o dado tem sua especificação distribuída ao longo das funções que implementam estas ações. Porém, numa análise da arquitetura do sistema, que se baseia não só na facilidade com que o sistema foi obtido , mas principalmente nas facilidades oferecidas ao longo do tempo, avalia-se um baixo desempenho de tal método. Principalmente, pelo fato dos métodos de funções estabelecerem uma forte dependência temporal na execução das funções.

Numa abordagem onde o sistema é baseado nos dados, a metodologia é denominada metodologia de objetos, ou melhor, ORIENTADA A OBJETOS. No caso, especifica-se um dado destacando-se suas características, propriedades e serviços disponíveis, definindo-se um módulo. No módulo, toda a implementação é escondida internamente e ele fica caracterizado pelos serviços que oferece.

Um sistema é projetado pela composição de módulos, onde os módulos relacionam-se entre si através dos serviços disponíveis.

Destacam-se dois aspectos importantes nesta metodologia de orientação a objetos :

1 - Não há uma dependência temporal entre os serviços disponíveis. Eles são definidos de acordo com as exigências do sistema e acrescidos ao módulo convenientemente, sem estabelecer nenhuma ordem de execução;

2 - Uma modificação na estrutura de um dado, induz a uma modi ficação apenas no módulo onde o mesmo está definido, já que os módulos se relacionam através dos serviços disponíveis por um dado e não através de sua implementação.

Um módulo deve procurar satisfazer certas características, tais como :

- Composição Modular : módulos, que compõem um sistema, devem ter a condição de serem combinados livremente com o objetivo de compor novos sistemas, bem diferentes daquele onde foram inicialmente propostos;

- Compreensão Modular : módulos devem ter um significado próprio, independente de outros módulos. No pior caso, poder-se-á dar uma olhada em alguns módulos vizinhos;

▸ Continuidade Modular : uma modificação na especificação acarreta a modificação em um ou alguns módulos.

Com o método de orientação a objetos, a pergunta que surge ao se iniciar um projeto de software é :

- "Quais são os módulos que irão compor o sistema ?"

A partir daí, novos módulos são definidos e combinados com outros já existentes.

A figura **FIG2.1** mostra um esquema geral da técnica de encapsulamento, onde um módulo é definido e posteriormente utilizado através de seus serviços disponíveis.

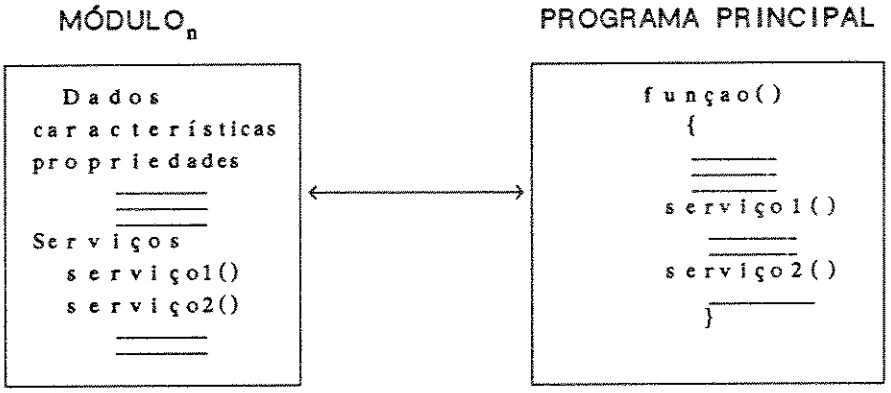


FIG2.1 - *Esquema Geral da técnica de Encapsulamento*

2.3.2. ABSTRAÇÃO DE DADOS

Termo em inglês : Data abstraction

A técnica de definir módulos agrupa dados com seus respectivos procedimentos tratando-os como elementos.

Suponhamos que se queira, em um sistema, definir dois elementos com as mesmas características. É necessário, portanto, um módulo gerenciador para manipular os diferentes elementos.

Este módulo gerenciador controla o uso dos serviços definidos no módulo por diferentes elementos.

Neste caso, o módulo não é tratado como um elemento único, mas sim, define um tipo que possui certas características e serviços, e o módulo gerenciador permite declarar vários elementos do mesmo tipo.

A figura **FIG2.2** ilustra um esquema geral de um sistema utilizando a técnica de abstração de dados.

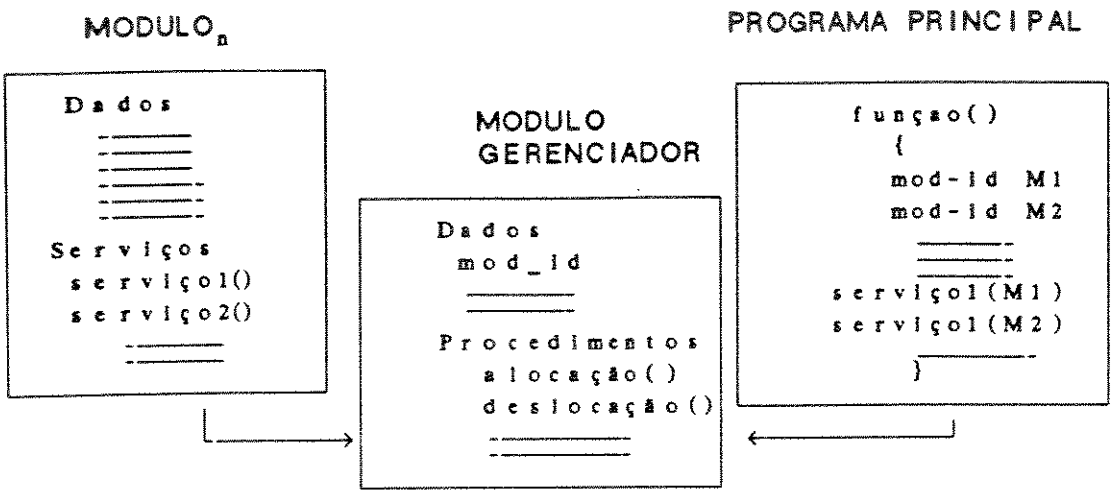


FIG2.2 - *Esquema Geral da técnica de Abstração de Dados*

As duas técnicas anteriormente mencionadas, Encapsulamento e Abstração de Dados, são muito utilizadas em linguagens como C e Pascal, com o objetivo de obter uma modularização do sistema que está sendo desenvolvido. Mas, apesar desta vantagem, as técnicas tornam-se muito artificiosas, pois há uma simulação de definição de tipos, tentando tratá-los como se fossem tipos básicos como int, char (já embutidos na linguagem) e portanto requer muita habilidade do programador para que estes módulos definidos desempenhem o papel de um tipo, como por exemplo, liberar o espaço reservado por um elemento do tipo definido, quando este sai fora de seu escopo. Isto é feito através do módulo gerenciador que reserva o espaço na memória e libera-o quando não é mais necessário como é visto na figura **FIG2.2**. Também, esta simulação de tipo acaba provocando ineficiência de execução e pouca clareza no sistema de software.

Quando uma linguagem suporta tais técnicas é possível encapsular a implementação de dados num módulo. Este módulo serve como um tipo para que diferentes elementos que satisfaçam as mesmas propriedades possam ser definidos e, o próprio compilador se encarrega de gerenciar os espaços necessários na memória para que os eslementos sejam manipulados.

O conceito de **TIPO ABSTRATO DE DADO** engloba as técnicas de encapsulamento e abstração de dados. Tem suporte para encapsular dados com seus serviços e gerenciar vários elementos do mesmo tipo. É um definidor de tipos especiais.

Portanto, abstração de dados consiste em tratar os módulos como definidores de classes de objetos que satisfazem as mesmas propriedades e têm disponível os mesmos serviços e, a partir disto, usar estas classes para declarar e inicializar um ou mais objetos para desempenharem serviços ao longo de um programa.

Certas linguagens possuem ferramentas próprias para definir classes, procurando suportar todas as exigências de um tipo básico.

No caso da linguagem C++ (que é a linguagem utilizada para desenvolver o presente estudo) faz-se uso do conceito de **class** e **struct** para desenvolver sistemas de software utilizando o conceito de **TIPO ABSTRATO DE DADO**.

Um exemplo clássico, visto na literatura de orientação a objetos, é a definição de pilha. Não importa se a pilha está crescendo de cima para baixo ou vice-versa, ou se é feito uso da definição de lista encadeada para implementá-la, o que interessa é

que há funções de **push()** e **pop()** que devem satisfazer a propriedade de uma pilha : *last_in_first_out*.

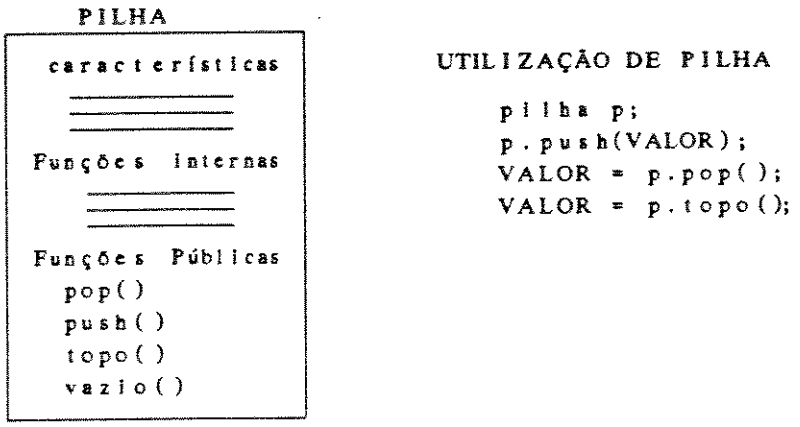


FIG2.3 - *Definição de pilha utilizando o conceito de TIPO ABSTRATO*

Observe no exemplo a utilização da PILHA. Foi descrito uma classe satisfazendo as propriedades que estão envolvidas no conceito de uma estrutura de pilha. No momento da utilização, a classe deve ser vista como um tipo, de tal forma que ao se declarar uma variável sendo de tal tipo, esta passe a ser um objeto contendo características, satisfazendo propriedades e executando as funções públicas de uma pilha.

Classe é o nome dado em substituição a tipo abstrato de dado, funções (ou procedimentos) públicas (os) em substituição aos serviços disponíveis de uma classe e objeto em substituição a uma instância de uma classe.

A definição de **TIPO ABSTRATO DE DADO** embutido numa linguagem é necessário, mas não suficiente, para que a mesma seja classificada como orientada por objeto. Outros suportes ainda são exigidos para satisfazer mais duas técnicas, que são : Mecanismo de Herança e Polimorfismo.

2.3.3. MECANISMO DE HERANÇA

Termo em inglês : inheritance mechanism

Normalmente, novos sistemas de software podem ser vistos como uma expansão de sistemas já desenvolvidos. Assim, uma forma adequada a seguir é desenvolver estes novos sistemas por imitação, refinamento ou combinação de classes já existentes que serviram para integrar um sistema já pronto.

Classes possuem qualidades esperadas de componentes de software reutilizáveis : são homogêneas, ou seja, segue-se os mesmos princípios básicos para defini-las e são coerentes, isto significa que cada classe é definida com um papel bem determinado, e suas interfaces são separadas de sua implementação pelo princípio de encapsulamento. Porém, mais características são necessárias para obter as metas de reutilização e

extensão.

No caso de reutilização, é necessário uma técnica que capture aspectos comuns entre classes com estruturas semelhantes, com o objetivo de evitar que o mesmo código seja escrito repetidamente, gastando-se tempo, introduzindo-se inconsistências e arriscando-se cair em erros.

Com relação à extensão, as classes garantem a consistência dos tipos no momento da compilação, mas não permitem a combinação de elementos de diversas formas.

Para uma evolução, tanto no aspecto de reutilização quanto de extensão, tem que se tirar vantagens das relações conceituais que ocorrem entre classes : uma classe pode ser uma EXTENSÃO, uma ESPECIALIZAÇÃO ou uma COMBINAÇÃO de outras classes.

Há necessidade de um suporte tanto a nível de método quanto de linguagem para se definir e se usar estas relações. O MECANISMO DE HERANÇA oferece este suporte.

Relacionamento de classes pelo princípio de EXTENSÃO é caracterizado pelo fato de uma classe **B** conter serviços entre os quais há aqueles que já estão definidos em outra classe **A**. No caso, **B** é declarada uma classe derivada de **A**, herdando os serviços disponíveis em **A** e implementando os serviços próprios.

A ESPECIALIZAÇÃO é o princípio de organizar classes com propriedades comuns, estabelecendo uma relação do tipo IS_A, utilizada na abordagem de redes semânticas [RIC83]. A figura **FIG2.4** ilustra um mecanismo de herança do tipo especialização no problema de figuras, exibido por Meyer [MEY88].

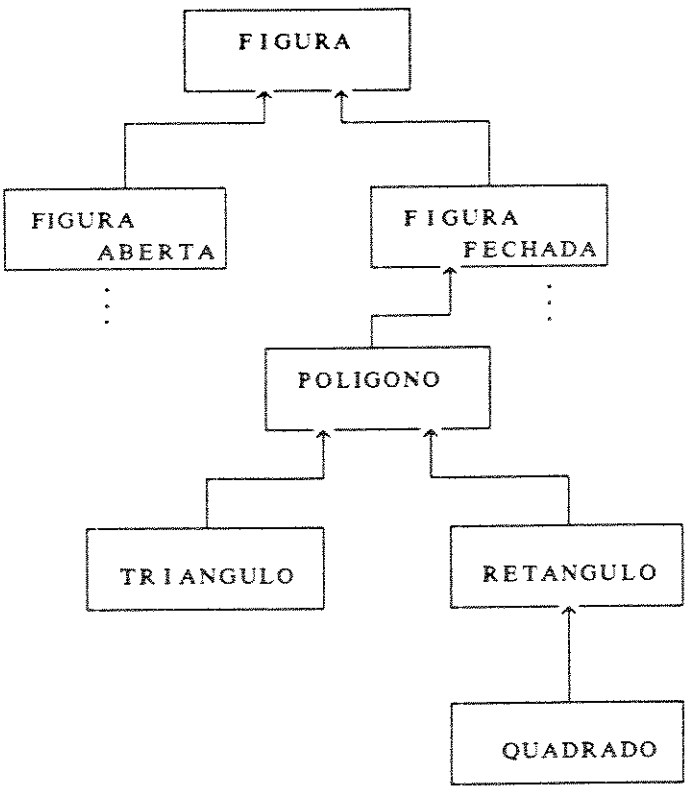


FIG2.4 - *Ilustração de um mecanismo de herança por especialização*

Na ilustração **FIG2.4** acima vemos que quadrado é uma especialização do retângulo, polígono uma especialização de figura fechada e assim por diante.

Através da herança múltipla é possível estabelecer uma COMBINAÇÃO de várias classes, ou melhor, uma combinação de propriedades e serviços.

Porém, o mecanismo de herança múltiplo acarreta o problema de repetição de nomes (funções diferentes em classes diferentes com o mesmo nome).

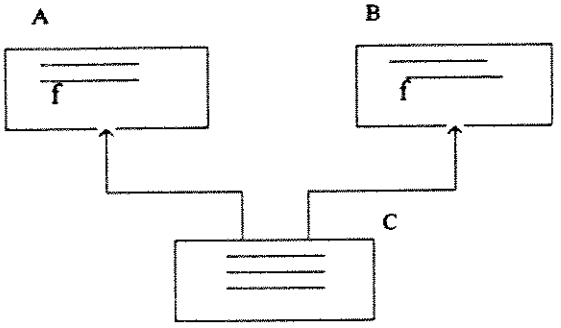


FIG2.5 - *Esquema de mecanismo de herança múltiplo com repetição de nomes em diferentes funções*

Há duas possíveis considerações para solucionar o problema :

- exigir que o cliente da classe **A** e **B** remova a ambiguidade;
- escolher uma interpretação "default" que tem como base um certo critério como, por exemplo, pela ordem em que as classes*_pai* estão listadas na classe **C**.

Há, ainda, classes que são definidas apenas com o objetivo de estabelecer um relacionamento entre classes derivadas, mas a classe em si não cria nenhum objeto. Tais classes são definidas como classes DIFERIDAS, pois serviços são especificados nesta classe, mas só são implementados em classes derivadas. O objetivo é facilitar a abstração de procedimentos.

Linguagens que suportam a definição de classes, mas não têm suporte para estabelecer MECANISMO DE HERANÇA, já não são consideradas ORIENTADA A OBJETOS. A linguagem C++ possui suporte para estabelecer MECANISMO DE HERANÇA entre classes.

2.3.4. Polimorfismo e Ligação Dinâmica

Termo em inglês : Polymorphism, Dinamic Binding

Como o próprio nome indica, polimorfismo é algo que se apresenta sob várias formas. No caso, é uma técnica de definir funções com o mesmo nome, mas que as implementações internas, apesar de terem o mesmo objetivo final, são descritas de formas diferentes.

Tal técnica permite que o programador desenvolva um sistema utilizando procedimentos de objetos sem se preocupar como eles são implementados. Esta técnica é bastan-

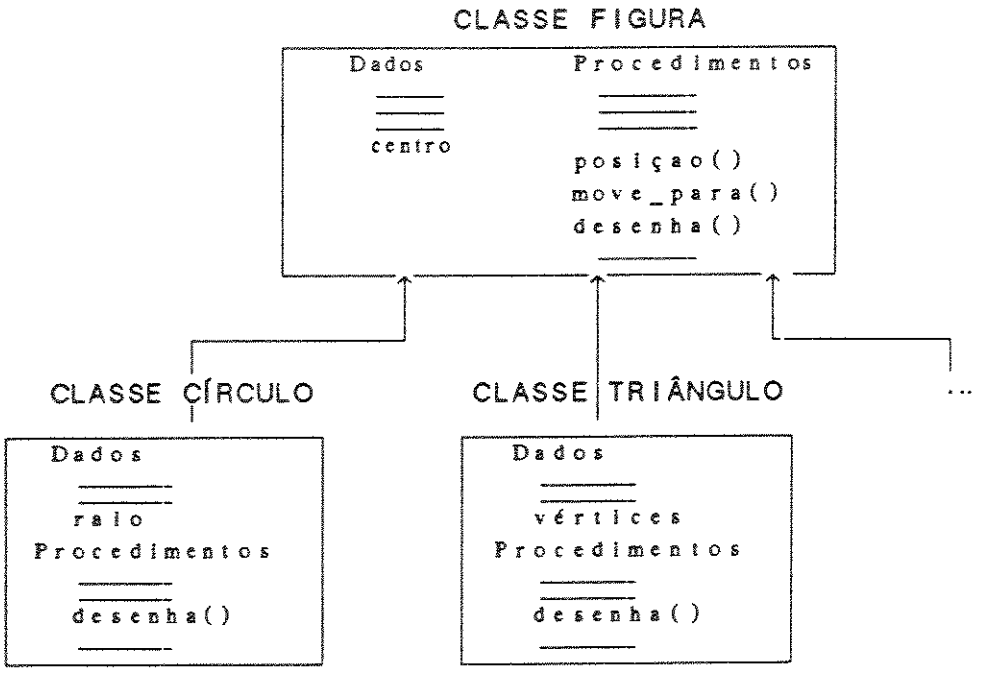
te significativa quando o mesmo tipo geral de ação pode ser realizado de diferentes modos por diferentes objetos.

Quando derivam-se classes de uma classe base, certos procedimentos precisam ser diferentes em cada classe. Por exemplo, considere a definição de figuras planas. Há as características e funções próprias de cada figura, como o procedimento de desenhar a figura (desenhar uma circunferência é diferente de desenhar um quadrado). Neste caso, deve ser possível definir a função desenha() em cada classe derivada.

No programa principal, quando um objeto ativa a função **desenha**(), o compilador identifica qual objeto está ativando o procedimento **desenha**() e assim faz a chamada correspondente. Este é um caso de polimorfismo com ligação adiantada ("early binding"), pois o procedimento que deve ser ativado é decidido no momento da compilação.

Suponhamos agora, que tem-se um vetor de apontadores para figu ras e queira-se desenhar todas elas. O procedimento que desenha todas as figuras, não sabe, exatamente, quais figuras estão referidas pelos apontadores contidos no vetor, mas a função desenha() é ativada para cada apontador contido no vetor. O procedimento é compilado, e só no momento de execução é que há a correspondência da função **desenha**() e a figura. Neste caso, o polimorfsmo é com ligação tardia ("late binding"), pois a função que deve ser ativada é decidida no momento de execução. Aqui, há uma abstração muito maior de programação, pois o programador pode definir um procedimento sem se preocupar quais figuras são desenhadas e de que modo, simplesmente preocupa-se que, num certo momento, algumas figuras devem ser desenhadas.

A figura **FIG2.6(a)** apresenta um classes definidas utilizando-se a técnica de polimorfismo para definir a função desenha(). A **FIG2.6(b)** utiliza-se objetos das classes definidas ilustrando um polimorfismo com ligação adiantada e ligação tardia.



- (a) Esquema de definição de figuras planas -

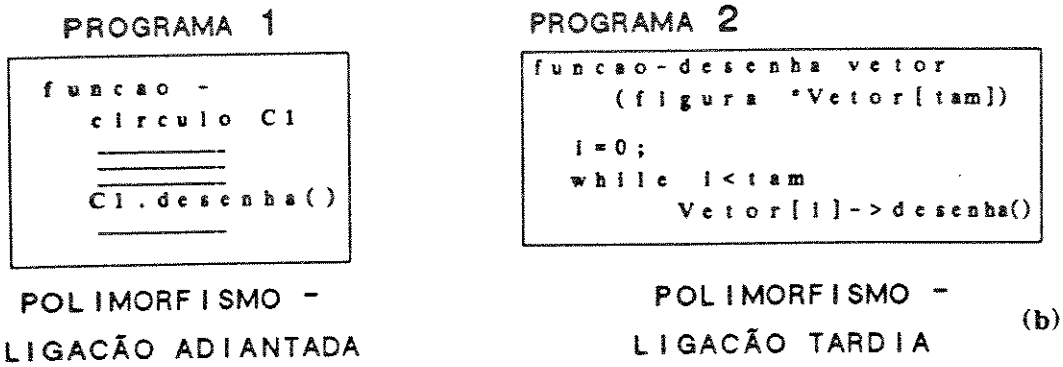


FIG2.6 - *Exemplos de polimorfismo com ligação adiantada e ligação tardia*

O **Programa 1**, quando compilado, identifica que **c1** trata-se de um círculo e faz a correspondência entre figura e função, já no **Programa 2**, há um vetor de ponteiros para figuras e a função percorre um a um, aplicando a função desenha sem saber quais são os tipos de figuras contidas no Vetor, e só no momento da execução que há a correspondência entre figura e função.

Observe que a técnica de herança,juntamente com a ligação dinâmica, permite uma arquitetura de software altamente descentralizada, onde toda variante de uma operação polimórfica é declarada dentro da classe que descreve a estrutura de dado variante correspondente.

2.4. LINGUAGENS ORIENTADA POR OBJETO

As técnicas descritas no tópico anterior são necessárias e suficientes para serem utilizadas no projeto e na implementação de um sistema segundo uma metodologia de orientação a objetos.

Uma pergunta que surge em iniciantes na metodologia é quais são os objetos que compõem o sistema. Segundo Meyer [MEY88], não há ainda um mecanismo adequado que responda a esta questão. Atualmente, definir quais os objetos que compõem um sistema se dá através de experiências absorvidas de outros trabalhos e de uma prática adquirida.

Paralelamente ao estudo do método de orientação a objetos, linguagens de programação estão sendo desenvolvidas para que possam suportá-lo. No que se segue são enumeradas algumas linguagens desenvolvidas [MEY88] :

1 - Linguagens com encapsulamento : Estas linguagens não suportam todas as técnicas, mas serviram de inspiração para o desenvolvimento de outras linguagens. São assim denominadas pois suportam basicamente a técnica de encapsulamento e/ou abstração.

MODULA_2 : projetada por Niklaus Wirth,

CLU : projeto do MIT sob direção de Barbara Liskov,

Ada : desenvolvida por um grupo do Departamento de Defesa US chefiado por Jean

Ichbiah

Mesa : projeto da Xerox.

2 - Linguagens orientada por objetos

SIMULA - desenvolvida em 1967 por Ole_Johan Dahl e Krysten Nygaard, Universidade de Oslo e Centro de Computação da Noruega,

SMALLTALK - desenvolvida pela Xerox, o projeto teve início em 1970 na Universidade de Utah por Alan Kay e Adele Goldberg.

Smalltalk não é somente uma linguagem, porém um ambiente de programação, cobrindo muitos aspectos direcionados para o sistema operacional.

O ambiente é caracterizado por uma interface com o usuário bastante sofisticada. Introduziu, ou melhor, popularizou muito dos avanços nesta área: múltiplas janelas, "icons", integração de textos e gráficos, menus e uso de "mouse" como um dispositivo para seleção e indicação.

Estas técnicas de interface são combinadas no ambiente Smalltalk com a técnica de orientação a objetos, tornando os objetos de um programa visíveis e acessíveis.

EXTENSÕES DE C

A linguagem C tornou-se nos últimos anos uma das linguagens mais difundidas no meio industrial e acadêmico. Portanto, dois esforços separados têm sido feitos com o objetivo de oferecer benefícios de orientação a objeto para programadores já treinados em C : C++ e Objective-C.

Apesar de terem um papel comum, as duas linguagens diferem-se significativamente. C++ mostra uma clara influência de Simula enquanto que Objective-C é um pré-processador com uma construção "Smalltalk_like".

C++ - projetada por Bjarne Stroustrup

É uma extensão de C com várias facilidades, as direcionadas para orientação a objetos e outras para melhoria da própria linguagem C.

Objective-C - projetada por Brad Cox.

EXTENSÕES DE LISP

Orientação a objetos tem despertado interesse na comunidade de Inteligência Artificial. A linguagem inicialmente escolhida na área de IA foi LISP, pelo menos nos meios acadêmicos.

A escolha desta linguagem é devido às técnicas oferecidas por um bom ambiente de LISP. Elas favorecem a aplicação direta de técnicas orientada a objetos. São técnicas, tais como : uma representação uniforme para programas e dados e "garbage collection".

Outras linguagens que destacam-se nesta categoria são : LOOPS (Xerox), FLAVORS (MIT) E CEYX (INRIA).

2.5. APRESENTAÇÃO GERAL DO TRABALHO DESENVOLVIDO

No trabalho, a linguagem C++ é utilizada na implementação das classes definidas abaixo. Ela oferece suporte para as técnicas discutidas anteriormente e outras facilidades que tornam a programação mais clara e eficiente. O apêndice A apresenta com detalhes todas as suas características.

As classes implementadas são definidas de forma relativamente independentes. O objetivo é integrá-las para mostrar a manipulação de dados, a nível de interface e armazenamento em arquivos, onde estes dados são representados previamente a partir de uma combinação adequada de tipos.

As principais classes implementadas com seus respectivos serviços disponíveis são as seguintes :

(1) CAMPO_MENS é uma classe que oferece serviços para manipulação de **campo_mensageiros**.

campo_mensageiro é o nome dado a uma estrutura interna que é criada pela função **campo_mensag()**.

A função **campo_mensag()** cria tipos para representar um conjunto de dados que possuem características e restrições próprias. Tais informações ficam internamente organizadas nos **campo_mensageiros**. Há 12 maneiras diferentes de se definir estes tipos, eles podem ser declarados como : String, Longo, Inteiro, Double, Vazio, Produto_Vetorial, Lista, Btree, Menu, União, Raiz e Endereço.

Os serviços disponíveis são :

(a) **grava_mens()** - grava em arquivo um **campo_mensageiro**,

(b) **carrega_campo()** - passa as informações contidas no **campo_mensageiro** para um objeto da classe CAMPO,

(c) **larg_mens()** - retorna o comprimento do **campo_mensageiro**.

(2) CAMPO é uma classe que oferece serviços para manipulação de dados do tipo estabelecido na inicialização de um objeto CAMPO. Os serviços disponíveis são :

(a) **campo**(*CM*) - declara um objeto CAMPO , inicializando parâmetros internos com as informações contidas no **campo_mensageiro** *CM*. A partir deste instante, o objeto CAMPO tem suporte para manipular com dados do tipo especificado por *CM*,

(b) **deft()** - faz com que o dado contido no objeto assuma seu valor 'default',

(c) **compara**(*Folha*) - compara o dado contido no objeto com o conteúdo em *Folha*,

(d) **verifica()** - verifica se o dado contido no objeto satisfaz as restrições impostas pelo *tipo* de dado explicitado na inicialização,

(e) **imprime()** - mostra na tela o dado atualmente no objeto,

(f) **edita()** - permite que um dado seja lido da tela e carregado para a memória numa região adequada do objeto. O dado é inserido através de uma interface que é esboçada utilizando um ambiente de edição. Esta interface é definida em conformidade ao tipo estabelecido previamente,

(g) **le()** - lê de uma certa posição do arquivo um dado que é carregado para a memória num lugar reservado pelo objeto CAMPO,

(h) **grava()** - grava em arquivo o dado atualmente no objeto,

(i) **caminho**(*CAM*) - em situações onde o objeto CAMPO está manipulando com dados compostos, este procedimento permite acessar elementos componentes do dado, através do caminho indicado por *CAM*,

(j) **uso()** - retorna o ponteiro para o dado atualmente contido no objeto CAMPO,

(h) **limpa_campo()** - limpa o espaço reservado, por um objeto CAMPO, para manipulação de dados,

(i) **compacta()** - compacta o dado atualmente no objeto para ser gravado em arquivo,

(j) **descompacta()** - descompacta o dado lido de um arquivo para ser disposto no espaço reservado por um objeto CAMPO,

(k) **limpo()** - verifica se o espaço, reservado pelo objeto CAMPO para manipulação de dados, está limpo ou não.

Todas estas funções contidas em CAMPO têm a característica de polimorfismo, pois dependendo do *tipo* de dado que está sendo manipulado, a função própria de cada um dos *tipos* é ativada adequadamente de uma classe derivada. São 12 classes derivadas de CAMPO de acordo com as maneiras possíveis de representar os dados.

Isto permite uma enorme flexibilidade diante da implementação de novos *tipos*, sem necessitar de nenhuma alteração na classe CAMPO.

A figura **FIG2.7** ilustra a relação da classe CAMPO com as classes derivadas. É utilizado um mecanismo de herança para estabelecer uma relação de especialização entre as classes. A classe CAMPO possui as características e funções que são independentes do tipo ao qual representa. Já as classes derivadas descrevem as características e funções específicas para cada tipo.

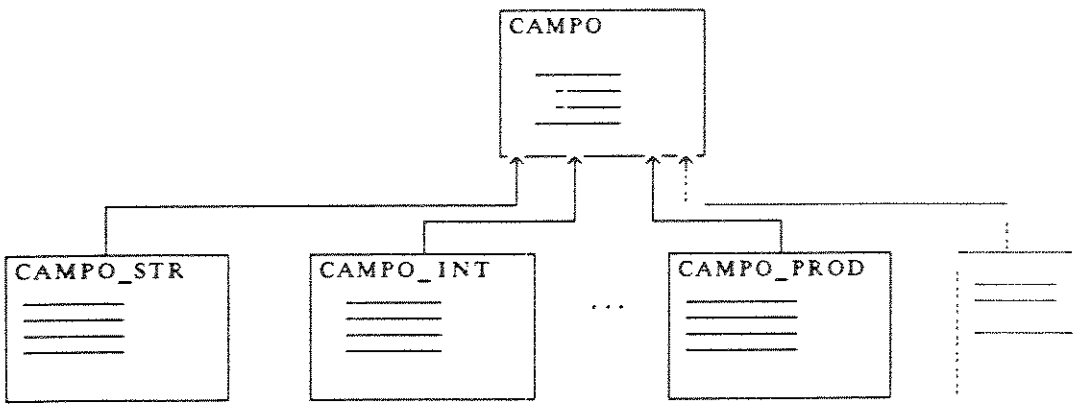


FIG2.7 - *Relacionamento da classe CAMPO com as classes derivadas*

(3) FICHÁRIO e REMENDO são classes cujos objetos manipulam com dados em disco, fazendo os gerenciamentos de espaços disponíveis. Ambas as classes são derivadas de uma classe base denominada DISCO. DISCO é uma classe DIFERIDA que estabelece um relacionamento entre as classes FICHÁRIO E REMENDO que têm objetivos comuns.

Os serviços disponíveis pelas duas classes são os mesmos. FICHÁRIO se difere de REMENDO pela maior simplicidade na sua estrutura e portanto, uma maior simplicidade com que dispõe os dados em arquivos.

Os serviços disponíveis pelas classes são :

(a) **remendo() (fichario())** - declara um objeto REMENDO (FICHÁRIO) e inicializa os parâmetros internos,

(b) **busca()** - retorna o dado armazenado no endereço indicado como variável da função,

(c) **insere()** - grava um dado no arquivo, no primeiro espaço disponível,

(d) **deleta()** - apaga um dado do arquivo, e reorganiza o arquivo de acordo com o espaço liberado,

(e) **~remendo() (~fichario())** - libera os espaços dinamicamente ocupados por um objeto REMENDO (FICHÁRIO).

(4) **BTREE** é uma classe cujos objetos organizam ordenadamente certos dados. O objeto funciona como uma árvore balanceada modelo ISAM (Index Search Access Method).
Os serviços disponíveis nesta classe são :

(a) **btree()** - declara um objeto BTREE e inicializa os parâmetros internos,

(b) **limpa_RAM()** - libera espaços reservados dinamicamente na memória,

(c) **seta_aspirante()** - atribui um valor na variável ASPIRANTE. ASPIRANTE é um espaço interno no objeto BTREE que guarda a chave de um dado qualquer que está sendo manipulado pelo objeto,

(d) **seta_dado()** - atribui um valor na variável DADO. DADO é um espaço interno no objeto BTREE que guarda o dado que está sendo manipulado pelo objeto,

(e) **insere()** - insere os conteúdos em ASPIRANTE e em DADO na btree,

(f) **busca()** - busca pelo dado que está sob a chave contida em ASPIRANTE,

(g) **deleta()** - apaga um dado que está sob a chave contida em ASPIRANTE,

(h) **anterior()** - posiciona no dado anterior da btree com relação ao dado atualmente posicionado,

(i) **posterior()** - posiciona no próximo dado da btree com relação ao dado atualmente posicionado.

A classe BTREE possui certas particularidades que são discutidas, com detalhes, no próximo capítulo.

(5) **MENU** é uma classe cujos objetos representam uma estrutura simples de menu e possui procedimentos para sua manipulação. Os serviços disponíveis são :

(a) **menu()** - declara um objeto MENU e inicializa seus parâmetros internos,

(b) **desenha()** - desenha o menu na tela,

(c) **escolha_menu()** - retorna a opção de escolha do menu,

(d) **executa_menu()** - gerencia as teclas de interrupção até que haja uma escolha no menu,

(e) **~menu()** - libera os espaços dinamicamente reservados por um objeto MENU.

(6) EDITOR é uma classe cujos objetos representam um ambiente de edição de textos com serviços disponíveis que o manipula.
Os serviços disponíveis são :

(a) **editor()** - declara um objeto EDITOR e inicializa os parâmetros internos,

(b) **manipula_editor()** - gerencia as teclas de interrupção de modo que seja possível editar um texto, podendo, eventualmente, gravá-lo em arquivo. Este gerenciamento permite apagar uma linha, posicionar no início ou no final do texto e apagar um caracter. Enfim, permite executar os comandos básicos de uma edição.

(7) INTERF é uma classe com serviços disponíveis para apresentação de janela para manipulação de dados.
Os serviços disponíveis são :

(a) **interf()** - declara um objeto INTERF e inicializa parâmetros internos. A inicialização se dá a partir de duas variáveis:
1 - um arquivo contendo o esboço de uma janela, desenhada no ambiente EDITOR+MENU e que será decodificada pelo objeto INTERF e
2 - um objeto CAMPO que manipula dados de um certo tipo.Os campos componentes deste objeto CAMPO (no caso do objeto estar manipulando com dados de um *tipo* composto) são distribuídos ao longo da janela.

(b) **imprime_janela()** - apresenta a janela decodificada na tela,

(c) **manipula_janela()** - acessa os campos dos dados que estão distribuídos ao longo da janela,

(d) **retorna_fundo()** - apaga a janela da tela, retornando o fundo,

(e) **~interf()** - libera os espaços dinamicamente ocupados pelo objeto INTERF.

Além destas classes citadas, que representam as classes principais da implementação, há outras classes entre as quais algumas são derivadas de classes bases, como é o caso de PARAM_BTREE derivada de BTREE. Ela representa uma estrutura dos parâmetros de um *tipo* BTREE, que serve para representar dados do tipo conjunto ordenado.

A figura **FIG2.8** estabelece o relacionamento das classes no sistema desenvolvido.

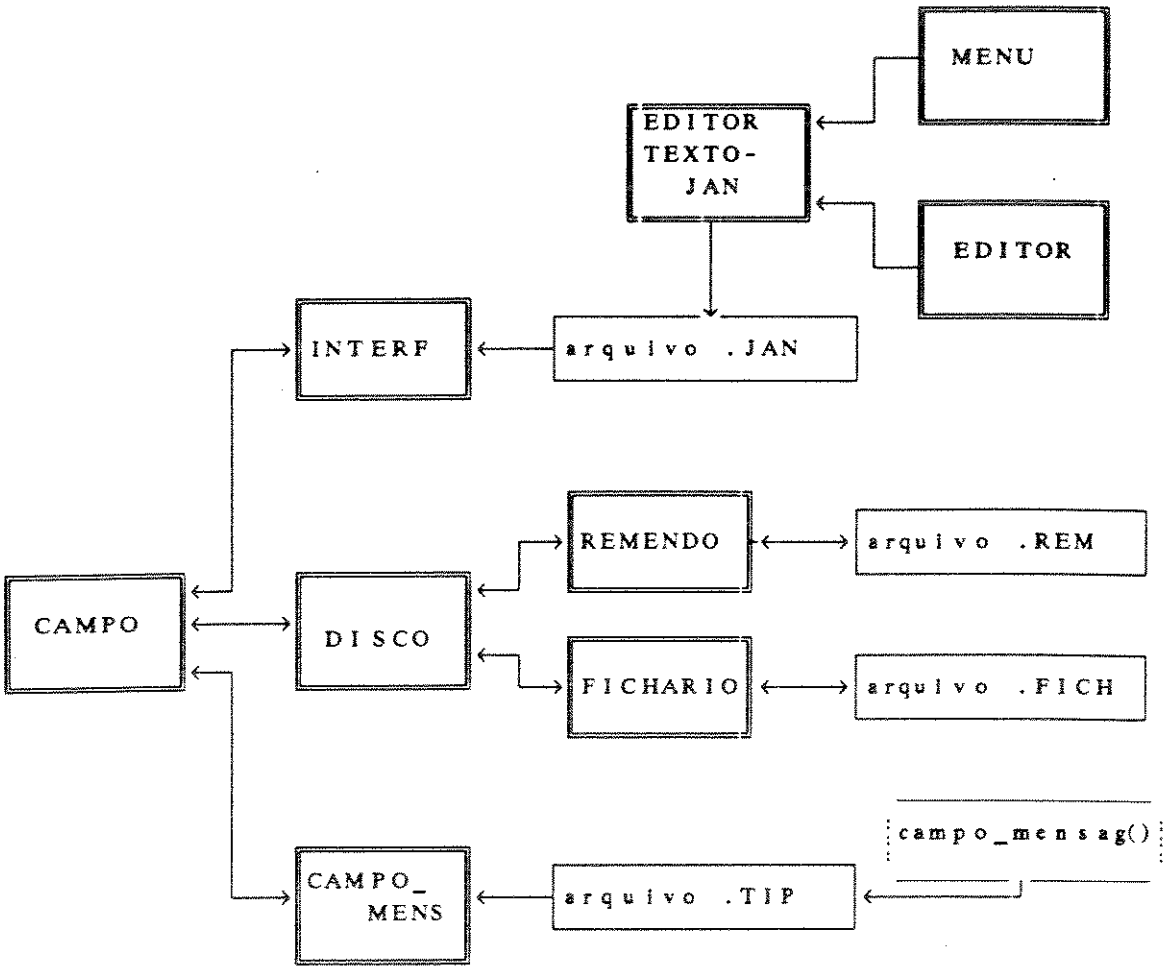


FIG2.8 - *Relacionamento da classe CAMPO com as demais classes*

Os blocos de contorno duplo indicam classes, os blocos de con torno simples indicam arquivos e o bloco de contorno pontilhado indica uma função simples.

O diagrama **FIG2.8** representa o seguinte comportamento :

1 - a função **campo_mens()** cria uma estrutura **campo_mensageiro**, onde esta contém as características e restrições de um tipo que irá representar uma certa "família" de dados,

2 - Paralelamente, num ambiente EDITOR_TEXTO_JAN, definido a partir de objetos das classes EDITOR e MENU, é esboçada uma janela para representar o tipo definido por **campo_mens()**. A janela é gravada num arquivo de extensão .JAN.

3 - Um objeto da classe CAMPO_MENS acessa o **campo_mensageiro** no arquivo e utiliza-o para inicialização do sistema. Um objeto INTERF decodifica a janela contida no arquivo de extensão .JAN, o qual também entra como variável na inicialização do sistema.

Se a estrutura para manipulação de arquivo é um objeto da classe FICHÁRIO o arquivo utilizado para armazenamento de dados é de extensão .FIC, mas se é um objeto da

classe REMENDO o arquivo é de extensão .REM.

Com a inicialização do sistema, um objeto CAMPO passa a manipular com dados do tipo caracterizado pelo campo_mensageiro fornecido na inicialização. A manipulação consiste na apresentação da interface, de maneira que dados possam ser inseridos e gravados em arquivo para uma eventual busca ou remoção posterior.

2.6. SUMÁRIO

As técnicas que caracterizam a metodologia de orientação a objeto para projetos de sistemas de software foram apresentadas. Posteriormente, linguagens que suportam tais técnicas foram enumeradas e finalmente houve uma apresentação geral do trabalho.

Todas as técnicas de orientação a objetos são exploradas no trabalho, com o objetivo de gerar um sistema modular e descentralizado e portanto sujeito a uma manutenção e reutilização de modo eficiente.

O polimorfismo é utilizado exaustivamente na classe CAMPO para que diferentes tipos de dados possam ter o mesmo tratamento. Portanto, como é inevitável, a extensão de novos tipos torna-se uma tarefa descentralizada, sem nenhuma necessidade de acessar as classes até então implementadas.

No próximo capítulo, o trabalho será apresentado em detalhes, destacando-se de que maneira as técnicas foram exploradas.

CAPÍTULO 3

UM MODELO PARA REPRESENTAÇÃO DE DADOS

3.1 - GENERALIDADES

Meyer [MEY88] discute as facilidades que a metodologia de orientação a objetos traz para o desenvolvimento de software nos aspectos relacionados à reutilização e extensão. Em particular, o presente trabalho visa destacar as qualidades de tal método aplicando-o no desenvolvimento de um sistema que manipula dados representados previamente.

A principal meta é estudar o problema da representação de dados que não possuem uma uniformidade, ou seja, que contenham informações que se expressam de formas distintas (como elementos atômicos, como produtos vetoriais, como conjuntos, como listas ou como condicionais). Com isto, o trabalho dedica-se na implementação de tipos capazes de representar estes dados, utilizando-se métodos de orientação a objetos para que os dados possam ser tratados uniformimente, independente do tipo que o represente ou de sua complexidade.

Aplicando-se o método, define-se e implementa-se classes que procuram satisfazer características como composição, compreensão e continuidade modular. Estas classes definem objetos bem caracterizados e com um papel bem definido, de modo que possam ser usados em contextos diferentes ao proposto.

Nos tópicos seguintes, são apresentados os módulos[1] que compõem o sistema implementado. Eles são os seguintes :

1 - CAMPOS : módulo onde são definidos 12 maneiras de especificação de tipos. Através de uma combinação adequada destes tipos é possível representar dados complexos como um elemento único.

O módulo é composto por 15 classes principais : 12 classes para cada tipo (todas derivadas de CAMPO) e as classes CAMPO, CAMPO_MENS, INTERF.

CAMPO_MENS é uma classe que manipula com um vetor, denominado **campo_mensageiro**, o qual contém as características e restrições de um tipo qualquer.

CAMPO é a classe que manipula com dados do tipo especificado na inicialização. É uma classe base para as 12 classes CAMPO_*tipo* derivadas.

INTERF é uma classe que faz a interface dos dados com o usuário.

2 - DISCO : módulo responsável pela manipulação de dados em arquivos. Esta manipulação consiste em gravar, ler e apagar dados de arquivos, fazendo-se um gerenciamento com relação ao espaço disponível.

O módulo é composto por 3 classes : DISCO, FICHÁRIO e REMENDO.

DISCO é uma classe base e FICHÁRIO e REMENDO são classes derivadas.

3 - BTREE : módulo responsável pela organização ordenada dos dados.

---

[1] módulo - neste contexto, um módulo é um conjunto de uma ou mais classes que estão relacionadas.

O módulo é constituído apenas pela classe BTREE. A classe BTREE comporta a estrutura de uma árvore balanceada modelo ISAM.

4 - EDIÇÃO : módulo responsável por gerar uma apresentação, por janelas, para os tipos previamente definidos. Este módulo inicializa um ambiente para desenhar janelas que são armazenadas, convenientemente, em arquivos .
O módulo é composto por 2 classes : MENU e EDITOR.

3.2. CAMPOS

Neste módulo concentra-se a parte principal da implementação realizada. Aqui são definidos os tipos que modelam dados.
Duas classes, CAMPO_MENS e CAMPO, tendo uma certa dependência mútua, são definidas com o intuito de se inicializar um tipo com suas características e restrições particulares e de manipular com dados de um certo tipo, previamente indicado.
Há 12 maneiras distintas para se definir tipos. Os tipos designam a forma como os dados devem ser tratados. Eles são inicializados com características e restrições próprias, de acordo com as exigências dos dados a serem representados. Portanto, o processo de inicialização de um tipo dota-o de significado, ou seja, de semântica.

Por exemplo, suponhamos que se deseje representar uma ficha contendo vários atributos a serem preenchidos, cada um com certas restrições próprias. Para isso, um tipo deve ser inicializado de maneira adequada para representar esta ficha.
Neste caso, a ficha seria uma n_upla (designada na implementação por um tipo Produto Vetorial), onde cada i_upla seria um tipo que se adequasse a cada atributo indicado na ficha. Suponhamos que um dos atributos fosse um código numérico que variasse dentro de um certo intervalo. Tal atributo seria representado por um tipo Inteiro compreendido entre o intervalo $[min,max]$ onde *min* representaria o valor mínimo e *max* o valor máximo possíveis ao atributo. Um outro atributo poderia ser um conjunto relativamente grande de valores numéricos reais. Este atributo já seria representado por um tipo Conjunto de tipo Double.
Assim, sucessivamente, é feita a inicialização de um tipo. Este tipo pode ser gravado, e toda vez que houver necessidade de manipular com dados que possam ser modelados por ele, é só acessá-lo através de um objeto manipulador de tipos, ou seja, um objeto da classe CAMPO.
Na inicialização de tipos é criado um vetor que irá conter todas as suas características. A este vetor é dado o nome de campo_mensageiro, e a classe CAMPO_MENS manipula com campo_mensageiros sem fazer distinção entre tipos.

Uma vez criado um tipo com características próprias, é possível utilizá-lo para inicializar um objeto da classe CAMPO.
A classe CAMPO, como já foi dito, é uma classe manipuladora de dados. Através desta classe é que dados, já com uma representação bem caracterizada, podem ser acessados.

A manipulação dos dados se reduz, por simplificação, ao armazenamento, leitura e remoção dos dados de arquivos, como também a apresentação dos mesmos em tela.
Esta apresentação do dado na tela é feita através de janelas que são desenhadas à parte por um objeto da classe EDIÇÃO (descrita no próximo tópico). Durante a apresentação, o usuário, com algumas teclas de interrupção, pode ativar as seguintes funções : atribuir valores 'default', gravar dado, atualmente disponível no objeto CAMPO, em arquivo, ler todos os dados guardados e apagar dados não mais necessários.

Portanto, juntamente com o campo_mensageiro designando o tipo do dado, na inicialização do sistema, é necessário especificar o arquivo onde os dados serão armazenados e o arquivo contendo a janela adequada para a interface.

A figura FIG3.1 apresenta um esquema de como inicializa-se o sistema para manipulação de dados.

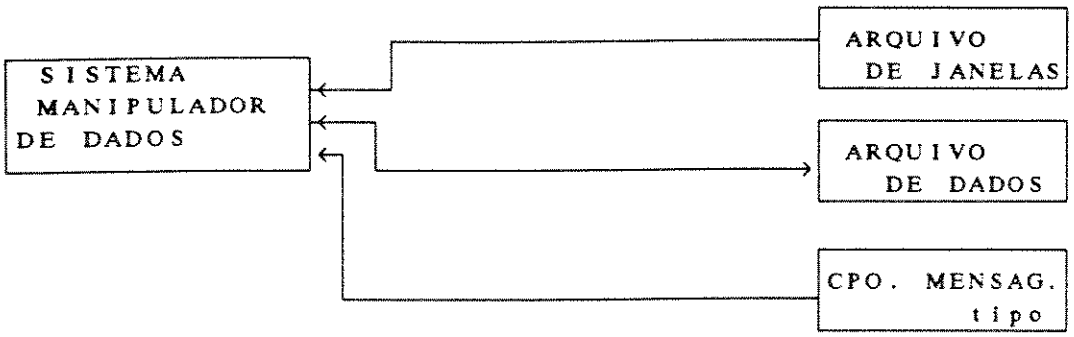


FIG3.1 - *Esquema de inicialização do sistema manipulador de dados*

Há 12 maneiras distintas disponíveis para inicializar um tipo para representar um certo dado, incluindo entre elas os tipos básicos já encontrados em C++ (INTEIRO, LONGO, DOUBLE e STRING). Tais tipos foram reconsiderados, pois além da definição conter certas particularidades, também há a necessidade de se ter uma coerência e uniformidade no tratamento dos tipos.

As maneiras de se definir os tipos são as seguintes:

1 - Inteiro
2 - Longo
3 - Double
4 - String
5 - Vazio
6 - Produto Vetorial
7 - Lista
8 - Btree
9 - Menu
10 - União
11 - Raiz
12 - Endereço

Classes foram implementadas para cada tipo declarado acima. Estas classes possuem funções próprias para manipulação dos dados de cada tipo.

Antes da descrição de cada tipo separadamente, o item seguinte ilustra , rapidamente, uma situação de utilização de um objeto CAMPO.

3.2.1. ILUSTRAÇÃO DA UTILIZAÇÃO DE UM OBJETO CAMPO

Considere uma ficha de cadastro de livros de uma biblioteca. Tal ficha é composta por alguns atributos, entre eles o nome do livro e autor.

O nome do livro é manipulado num atributo do tipo STRING. Este tipo é encontrado em linguagens de programação (no caso, C++) como um tipo básico, onde já existe uma

biblioteca de funções para manipulá-lo. O mesmo ocorre com os tipos INTEIRO, LONGO e DOUBLE. A princípio, não haveria necessidade de criar tal tipo.

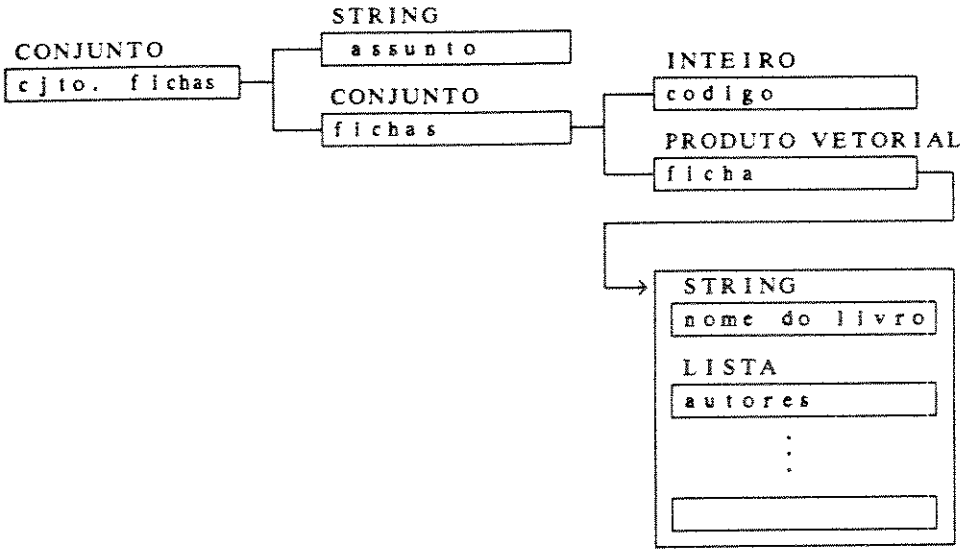
Observemos agora o atributo autor. Um determinado livro pode conter um ou mais autores. Desta forma, não é adequado manipulá-lo num atributo do tipo STRING, mas sim numa LISTA de STRINGS. Tendo-se um TIPO LISTA contendo funções adequadas e declarando tal atributo como um TIPO LISTA de STRINGS, há uma correspondência mais direta entre a informação a ser representada e a linguagem de representação.

Prosseguindo com o exemplo, suponhamos agora que estas fichas também estejam cadastradas por assunto. As fichas são como n_uplas de tipos, um tipo para cada atributo. Definir um TIPO PRODUTO VETORIAL para representar as fichas parece adequado.

Sob cada assunto há uma quantidade muito grande de fichas, onde um TIPO LISTA é inadequado pela ineficiência de manipulação. O adequado é um TIPO BTREE, onde as fichas são guardadas de forma ordenada permitindo um acesso eficiente a elas.

Finalmente, como numa biblioteca o número de assuntos também é grande, estes conjuntos de fichas podem ser ordenados por assunto sob um outro TIPO BTREE.

Uma vez criado tal *tipo* complexo, como esquematizado na figura **FIG3.2**, um objeto CAMPO pode ser inicializado com este *tipo* e passar a tratar dados deste *tipo* como elemento único, ou melhor, pode atribuir funções aos dados tratando a representação de forma global.



CMB - Campo mensageiro da Biblioteca
- TIPO DEFINIDO PARA REPRESENTAR UMA BILIOTECA -

```
campo BIBLIOTECA(CMB);
BIBLIOTECA.edita();
void *U = BIBLIOTECA.uso();
campo *PC = BIBLIOTECA.caminho("0/1/1/1");
BIBLIOTECA.imprime();
BIBLIOTECA.retorna_edita();
          :
          :
          :
```

- Algumas funções disponíveis pela classe CAMPO -

FIG3.2 - *Consideração hipotética de uma biblioteca com sua representação através de tipo*

3.2.2. DESCRIÇÃO DOS TIPOS

Quando um TIPO é definido, certas informações são exigidas para que ele possa ser bem caracterizado. Estas informações são colocadas numa sequência rígida e passa a constituir o **campo_mansageiro** do tipo definido.

Não importa qual o tipo que está sendo definido, a estrutura geral do **campo_mensageiro** é sempre a mesma. Há funções para manipulação dos **campo_mensageiros** e estas estão contidas na classe CAMPO_MENS, que contém o ponteiro para o **campo_mensageiro**, e funções como gravar e ler no/do arquivo o **campo_mensageiro**.

A seguir são descritas as características que devem estar contidas em cada **campo_mensageiro** para se definir um TIPO. Os exemplos apresentados, juntamente com a descrição, adotam uma notação simplificada para não se sobrecarregarem com especificações.

(TI) Tipo Inteiro :

Deve conter as seguintes características e restrições :
- Valor MÁXIMO permitido no atributo,
- Valor MÍNIMO permitido no atributo,
- Valor 'default' para o caso de omissão de dado.

(TL) Tipo Longo :

Deve conter as seguintes caracterísitcas e restrições :
- Valor MÁXIMO permitido no atributo,
- Valor MÍNIMO permitido no atributo,
- Valor 'default' para o caso de omissão de dado.

(TD) Tipo Double :

Deve conter as seguintes caracterísitcas e restrições :
- Valor MÁXIMO permitido no atributo,
- VAlor MÍNIMO permitido no atributo,
- Número de casas decimais após a vírgula,
- Valor 'default' para o caso de omissão de dado.

(TS) Tipo String :

Deve conter as seguintes características e restrições :
- Comprimento MÁXIMO permitido,
- Valor 'default' para o caso de omissão de dado.

(TV) Tipo Vazio :

É um tipo trivial. Não possui nenhuma característica interna e todas as funções relacionadas com este tipo são triviais ou nulas.

Há situações em que se guardam os dados como chaves numa btree. Portanto, o tipo do dado não necessita de uma especificação ao se inicializar um objeto BTREE (descrito no item 3.4). Neste caso, o tipo Vazio é a maneira adequada de declarar o tipo do dado na inicialização de um objeto BTREE.

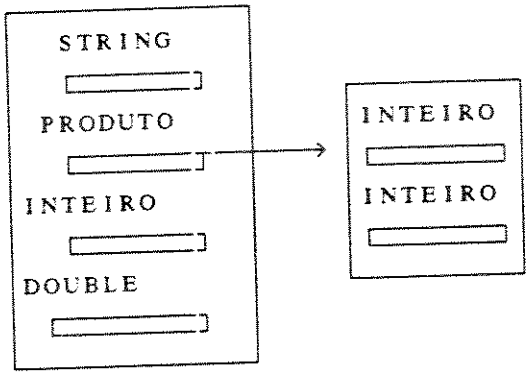
Outra situação em que utiliza-se o tipo Vazio é na especificação do tipo União, que será ilustrado quando o mesmo for descrito.

(TP) Tipo Produto Vetorial

É um tipo que representa uma n_upla de tipos quaisquer.

É utilizado para inicializar atributos compostos por vários sub_atributos, cada um tendo um tipo específico.

Deve conter as seguintes características :
- Número de **campo_mensageiros** que compõem a n_upla;
- **campo_mensageiros** que compõem a n_upla : descrição de cada tipo que compõe o tipo Produto Vetorial.




```
campo_mens *CM = campo_mensag(Produto,4,
                    campo_mensag(String,25,'campo0'),
                    campo_mensag(Produto,2,
                         campo_mensag(Inteiro,132,-4,17),
                         campo_mensag(Inteiro,1794,-25,0)),
                    campo_mensag(Inteiro,99,21,35),
                    campo_mensag(Double,2,132734.24,7.34,8.17));
```

FIG3.3 - *Exemplo de declaração de um tipo Produto Vetorial*

No exemplo acima, um **campo_mensageiro**, denominado CM, é criado e contém caracte-

rísticas de um tipo Produto Vetorial composto por 4 tipos :

1 - Tipo String com largura máxima de 25 e valor 'default' "campo0";

2 - Tipo Produto Vetorial composto por 2 tipos :

2.1 - Tipo Inteiro com valor máximo de 132, valor mínimo -4 e valor 'default' 17;

2.2 - Tipo Inteiro com valor máximo 1794, valor mínimo -25 e valor 'default' 0;

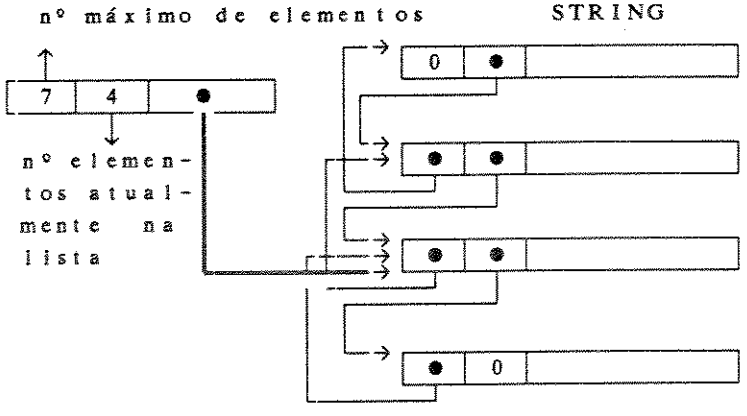
3 - Tipo Inteiro com valor máximo 99, valor mínimo 21 e valor 'default' 35;

4 - Tipo Double com valor máximo 132734.24 valor mínimo 7.34 e e valor 'default' 8.17.

(TL) Tipo Lista

É um tipo utilizado para inicializar um atributo lista de um tipo qualquer.
Deve conter as seguintes características :

- Número máximo de dados permitidos na lista;
- **campo_mensageiro** que compõe a lista : descrição do tipo de elemento que estará contido na lista, ou seja, se a lista é de INTEIRO, PRODUTO VETORIAL, STRING , LISTA e etc.



-Disposição interna de um atributo do tipo Lista-

```
campo_mens*CML = campo_mensag(Lista,7,
                        campo_mensag(String,40,"Especial"));
```

FIG3.4 - *Exemplo de declaração de um tipo Lista*

No exemplo FIG3.4 um **campo_mensageiro**, denominado CML, é criado e contém caracte rísticas de um tipo Lista com no máximo 7 elementos. Estes elementos são do tipo

String de comprimento máximo 40 caracteres e valor 'default' o 'string' "Especial".

Dentro do tipo Lista há um contador do número de elementos contidos na lista e um apontador para o elemento atualmente posicionado na Lista.

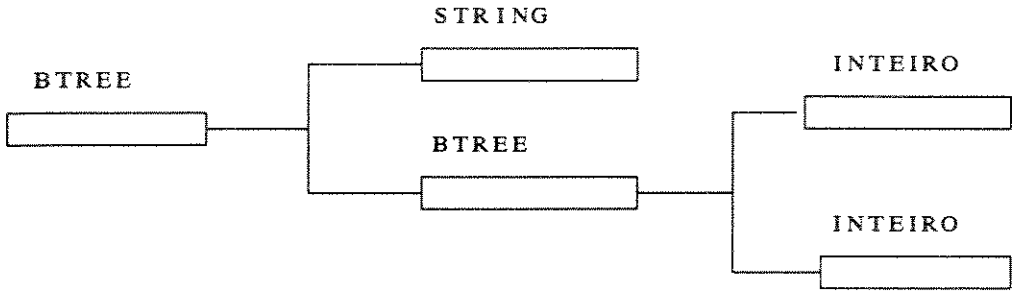
Quando um dado de um atributo tipo Lista é gravado em arquivo, antes ele é compactado adequadamete, constando apenas do número de dados dispostos na lista seguido de seus respectivos conteúdos, também compactados adequadamente segundo convenções do tipo que compõe o tipo Lista.

(TB) Tipo Btree

É um tipo que inicializa atributos que são conjuntos de um tipo qualquer.

Como este tipo é derivado da classe BTREE (descrita no item 3.4), ele deve conter todos os dados para inicialização de um objeto BTREE, que são :

- Número MÁXIMO de elementos dispostos numa folha base,
- Número MÁXIMO de índices dispostos numa folha índice,
- **campo_mensageiro** do campo CHAVE da btree,
- **campo_mensageiro** do campo DADO da btree.




```
campo_mens *CMB = campo_mensag(Btree,9,3,
                        campo_mensag(String,30,'Chave'),
                        campo_mensag(Btree,9,3,
                                campo_mensag(Inteiro,37000,-3),
                                campo_mensag(String,35,"Dado")));
```

FIG3.5 - *Exemplo de declaração de um tipo Btree*

No exemplo **FIG3.5**, um **campo_mensageiro**, denominado CMB, é criado contendo características de um tipo do tipo Btree e tem como chave o tipo String e como dado o tipo Btree que por sua vez tem como chave um tipo Inteiro e como dado um tipo String. No caso, o tipo está representando um conjunto cujos elementos são conjuntos.

O exemplo ilustra a possibilidade que a implementação da btree oferece para declarar conjuntos de conjuntos através da caracterização de um ponteiro de btree (PBTREE) (descrito no item 3.4).

(TM) Tipo Menu

Este tipo é especificado com o objetivo de inicializar atributos que aceitam certos tipos de dados bem determinados e em número pequeno. Um exemplo típico de dado que pode ser manipulado num atributo deste tipo é SEXO (MASCULINO/FEMININO).

As características que ele deve conter são :
- Número de escolhas do MENU,
- Vetor contendo as escolhas,
- Escolha 'default' do menu.

(TU) Tipo União

É um tipo adequado para inicializar atributos que manipulam com informações condicionais. Ele contém vários tipos diferentes simultaneamente, porém estes tipos são exclusivos entre si. De acordo com a instância contida num tipo Menu , necessariamente vinculado a ele, é que se decide qual dos tipos é o ativo para esta instância.

Deve conter as seguintes características :
- Número de **campo_mensageiros** que compõem a união;
- **campo_mensageiros** que compõem a união: descrição de cada tipo que está compondo a união.
- Caminho até ao atributo que está vinculado ao atributo do tipo União. Através deste atributo vinculado é que se decide qual deve ser o tipo de atributo ativo entre os que compõem a União.

A figura **FIG3.6** ilustra de forma genérica o tipo União.

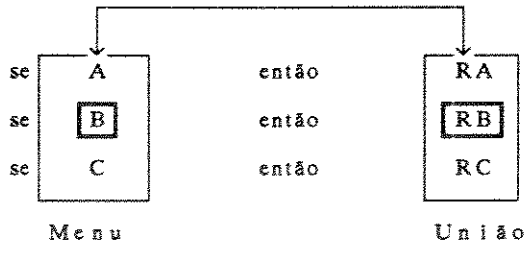


FIG3.6 - *Ilustração genérica de um tipo União*

No exemplo **FIG3.6** o atributo do tipo Menu possui 3 opções : A, B e C. No caso, a opção B está selecionada, fazendo com que o atributo do tipo União, vinculado a este, tenha ativo o tipo correspondente que é RB.

Pode existir situações em que dependendo da escolha de uma condicional não há nenhum atributo ligado a ela. Neste caso, o tipo Vazio é a especificação do atributo de tal condicionamento.

(TR) Tipo Raiz

Os 10 tipos, descritos anteriormente, representam dados bastante elaborados, bastando, para isso, combiná-los adequadamente como indicado pelos exemplos.

Este tipo e o próximo descrito funcionam como tipos internos, com o objetivo de organizar os tipos definidos com relação ao armazenamento em disco.

Depois de definido um tipo (combinando as 10 maneiras possíveis) utiliza-se o tipo Raiz, que deve conter o tipo previamente definido.

Um objeto CAMPO do tipo Raiz possui procedimentos que: grava o conteúdo do tipo *filho* num lugar fixo do arquivo e faz atualizações posteriores, como ler do arquivo e atualizar o conteúdo.

Este lugar fixo, geralmente, é o início de um arquivo, e o conteúdo deve ser, necessariamente, de um comprimento também fixo. Isto é para se definir, precisamente, a partir de que endereço o arquivo está disponível para inserção de outros dados.

Este tipo possui como característica o **campo_mensageiro** que tem seu conteúdo gravado num lugar fixo do arquivo.

(TE) Tipo Endereço

Este tipo, também, auxilia o armazenamento de dados em arquivo. É utilizado nos seguintes casos :

1 - O dado que está sendo armazenado em disco não tem um comprimento fixo : por exemplo, dados do tipo lista podem aumentar ou diminuir de tamanho quando insere-se ou apaga-se um elemento da lista. Deste modo, quando o dado é atualizado no arquivo, ele pode ocupar lugares diferentes. Portanto, o endereço da posição onde o mesmo é gravado, que é sempre de um tamanho fixo (longo), é que deve ser o conteúdo do tipo Endereço. Deste modo, quando grava-se uma lista, primeiro define-se um tipo Raiz que deve ter como tipo *filho* o tipo Endereço que por sua vez controla o conteúdo do tipo Lista.

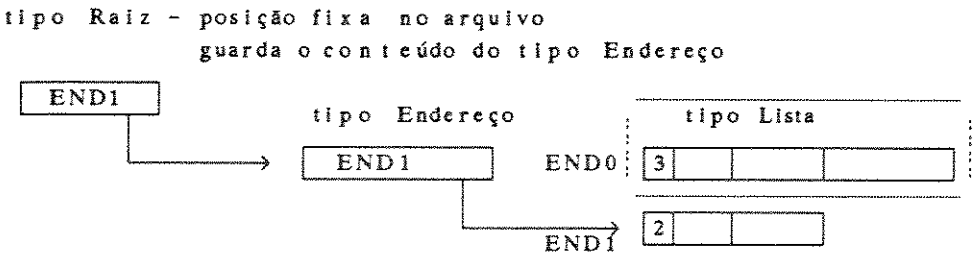


FIG3.7 - *Forma de armazenar um dado tipo Lista*

No exemplo **FIG3.7** o conteúdo no atributo do tipo Lista possuia 3 elementos e estava gravado em End0, depois passou a ter 2 elementos e foi gravado em End1. Com isso o tipo Endereço teve seu conteúdo modificado de End0 para End1 e consequentemente o tipo Raiz atualizou o endereço, na posição do arquivo na qual ele tem controle.

2 - O dado armazenado ocupa mais que 512 caracteres : quando o dado é do tipo composto, sendo representado pela combinação de vários tipos, deve-se tomar o cuidado para que o tamanho total do dado não ultrapasse 512 caracteres. Isto porque, as estruturas desenvolvidas para manipulação de dados em disco têm controle sobre dados de até 512 caracteres de comprimento. Quando este valor é ultrapassado, o dado deve ser desmembrado em dados menores e armazenados ao longo do arquivo. Isto se dá utilizando-se o tipo Endereço.

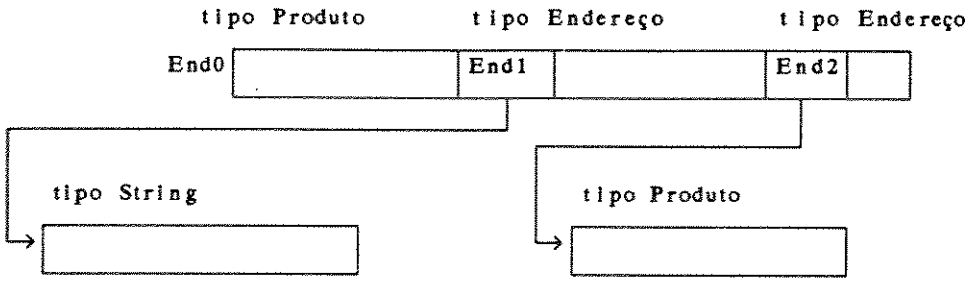


FIG3.8 - *Forma de armazenamento utilizando tipo Endereço*

No exemplo **FIG3.8** um dado do tipo Produto está gravado na posição End0 do arquivo, mas possui dois dados componentes gravados nas posições End1 e End2, que são controlados por dois atributos do tipo Endereço (um controlando um tipo String e outro controlando um outro tipo Produto), dispostos ao longo deste tipo Produto.

A tarefa de oferecer mais um tipo é feita sem alterar nenhum tipo já desenvolvido, bastando para isso acrescentar uma nova classe derivada de CAMPO redefinindo, convenientemente, os procedimentos adequados para o novo tipo.

3.3. DISCO

DISCO é o nome dado a uma classe genérica de estruturas para manipulação de arquivos.

Esta classe apenas especifica as funções públicas de uma estrutura de acesso a arquivos e que são redefinidas em classes derivadas, ou seja, trata-se de uma classe DIFERIDA que apenas relaciona classes que têm papéis em comum.

As funções redefinidas são GRAVA, REGRAVA, LÊ e REMOVE. Para que estas funções possam ser executadas há a necessidade de uma classe derivada, onde nesta classe as características da estrutura ficam estabelecidas. Isto porque, a classe DISCO não caracteriza nenhuma estrutura em particular, ela apenas declara 4 funções virtuais, ou seja, 4 funções que são redefinidas de acordo com as características da estrutura que as suportam, permitindo portanto formas diferentes de armazenamento em disco.

Foram desenvolvidas duas classes derivadas da classe DISCO para mostrar a generalização que esta oferece.

As classes derivadas são : FICHÁRIO e REMENDO.

A figura **FIG3.9** apresenta um esquema da relação entre estas classes.

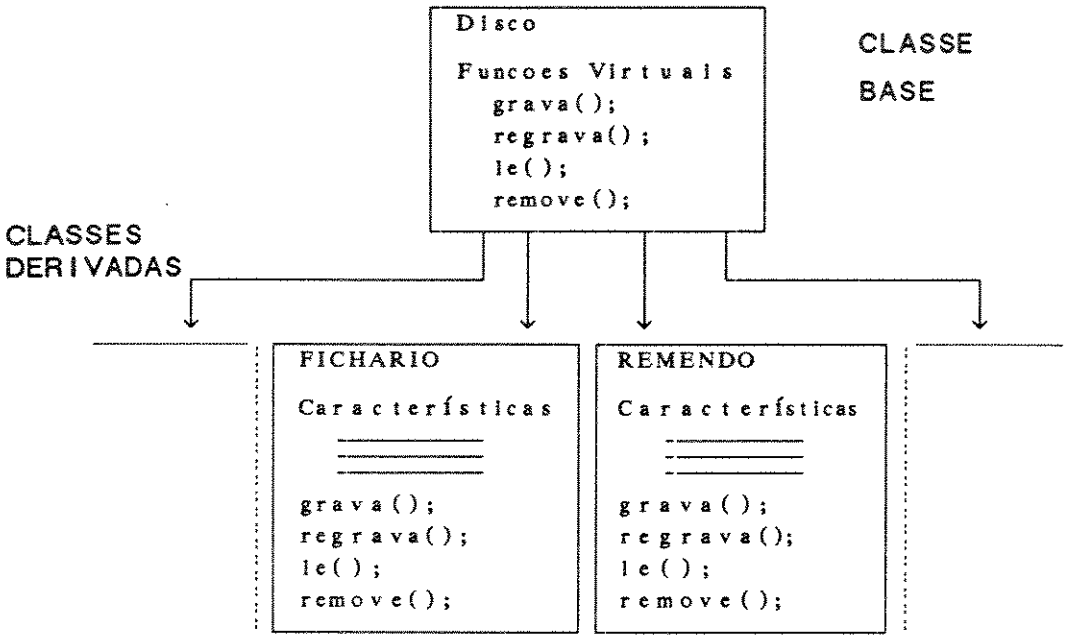


FIG3.9 - *Esquema de relacionamento da classe DISCO*

As linhas tracejadas indicam que novas classes podem ser redefinidas e acrescidas sem qualquer alteração nas classes já existentes.

3.3.1. FICHÁRIO

FICHÁRIO é uma classe, derivada de DISCO, cujos objetos suportam uma estrutura simples formada por fichas de largura fixa.

Um objeto do tipo FICHÁRIO possui o nome do arquivo onde são armazenados os dados e a largura das fichas que compõem o arquivo. As funções públicas são executadas da seguinte maneira:

(F1) GRAVAR : o objeto FICHÁRIO posiciona no final do arquivo e grava o dado ocupando uma ficha. Se o dado é menor que a largura da ficha, há "lixo" na parte não ocupada ou se o dado é maior, o mesmo é truncado;

(F2) REGRAVAR : o dado é regravado usando o mesmo procedimento anterior, só que numa ficha já existente indicada como parâmetro;

(F3) LER : a ficha indicada como parâmetro é carregada para a memória RAM;

(F4) REMOVER : não é feito nenhum procedimento para remover a ficha indicada. Simplesmente a ficha é esquecida e nunca se tem acesso à mesma para acrescentar outro dado.

Como descrito acima, um objeto da classe FICHÁRIO é caracterizado pela eficiência de tempo de execução sem se preocupar com o espaço ocupado, desnecessariamente, no arquivo.

Objetos desta classe são ideais em situações onde os dados devem ser gravados rapidamente e permanecem gravados por pouco tempo.

Esta classe permite que um arquivo comporte uma única estrutura FICHÁRIO.

3.3.2. REMENDO

REMENDO é uma classe , também derivada de DISCO, cujos objetos suportam uma estrutura mais elaborada. Permite o armazenamento de dados de comprimentos variados, entre 2 e 512 bytes e permite que até 10 (dez) estruturas REMENDO distintas compartilhem o mesmo arquivo.

A classe REMENDO ocupa-se da otimização do espaço utilizado. Quando um dado é inserido, ele ocupa exatamente o seu comprimento e quando um dado é removido, há uma compactação dos dados na estrutura.

A implementação da classe REMENDO, assim como de outras classes, faz uso de certos conceitos embutidos na linguagem C++ para tornar a implementação mais legível e eficiente :

1 - função OVERLOAD : há duas funções **insere()** e duas funções **deleta()**, uma para inserir (remover) "strings" e a outra para inserir (remover) dados com comprimentos estabelecidos. Este conceito permite uma coerência na utilização das funções. Apesar de haver algumas particularidades no dado que está sendo inserido, por exemplo, no final a ação é de gravar o dado no arquivo. Portanto, ambas as funções de inserção devem receber o mesmo nome;

2 - função DESTRUIDOR ('destructor') : há uma função que libera espaços dinamicamente ocupados na memória, por estruturas REMENDO, quando estas não são mais necessárias;

3 - funções PRIVADAS : há funções internas, utilizadas somente por outras funções da classe. Os usuários da classe nunca fazem uso de tais funções.

Tendo-se classes derivadas da classe DISCO torna-se possível fazer uso de uma estrutura de arquivo sem se preocupar com a estrutura em si (característica de polimorfismo).

Se um objeto de alguma classe faz uso de um objeto DISCO, este objeto pode manipular tanto os dados numa estrutura FICHÁRIO quanto numa estrutura REMENDO, sem necessitar de nenhuma especificação interna na implementação. Basta inicializar o objeto com um dos objetos de classes derivadas de DISCO.

No tópico seguinte, durante a descrição do módulo BTREE, há um esquema que ilustra a utilização da estrutura DISCO.

3.4. BTREE

BTREE é uma classe, cujos objetos suportam uma estrutura para organização de dados. Dados são armazenados sob chaves numa ordem estabelecida pela função de comparação contida no tipo da chave.

Os algoritmos das funções para manipulação de dados (tais como insere, busca e deleta) são os algoritmos padrões de uma árvore balanceada modelo ISAM.

As seguintes particularidades são acrescidas à classe BTREE :

1 - ORGANIZAÇÃO DE DADOS DE DIFERENTES TIPOS : ao inicializar um objeto da classe

BTREE, o usuário escolhe, de acordo com os dados que ele deseja guardar, quais são os tipos da chave e do dado. Como cada tipo definido possui funções próprias, a função **compara()**, que estabelece a ordem dos dados na btree, é lida do tipo escolhido como chave e passada para o objeto BTREE.

Este procedimento utiliza-se da técnica de polimorfismo, pois a implementação da função **compara()** é própria de cada tipo e o procedimento da classe BTREE, que faz uso desta função, preocupa-se com a ação que deve ser executada sem se preocupar como a ação foi implementada.

Os tipos possíveis de se atribuir à chave e ao dado são os definidos no item 3.2.

2 - DIFERENTES FORMAS DE ARMAZENAMENTO EM DISCO : outro parâmetro para inicialização de um objeto da classe BTREE é um ponteiro para um objeto da classe DISCO, pois a btree necessita de uma estrutura que manipule seus dados em arquivo.

Como já foi especificado, DISCO é uma classe genérica da qual derivam-se outras classes mais específicas. Assim, de acordo com a adequação, este parâmetro pode ser um objeto da classe FICHÁRIO, REMENDO ou outro objeto de uma classe, derivada de DISCO, que venha a ser definida. Desta forma, as funções utilizadas pelo objeto BTREE são executadas de acordo com o objeto DISCO tomado como parâmetro. Aqui, novamente, a técnica de polimorfismo é utilizada.

A figura **FIG3.10** ilustra as duas particularidades da classe BTREE descritas acima. As setas indicam as variações permitidas a um objeto da classe BTREE sem a necessidade de modificação da própria classe.

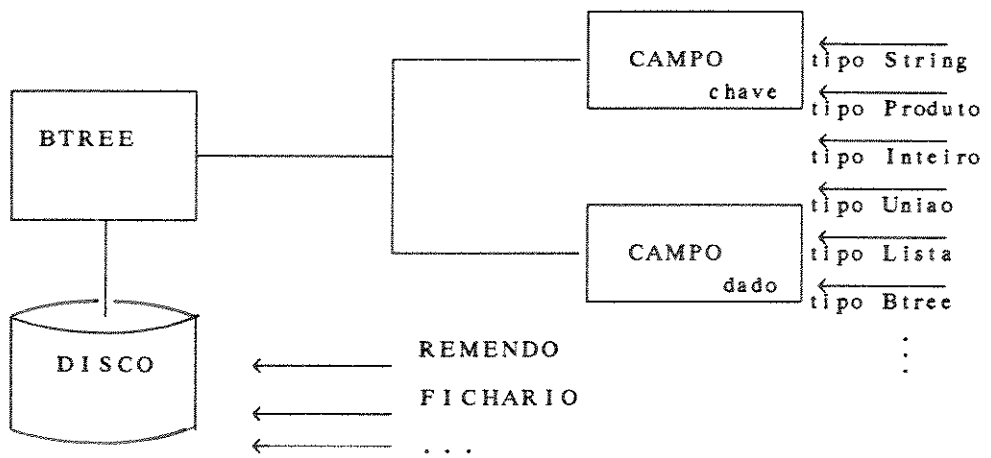


FIG3.10 - *Esquema das flexibilidades permitidas a um objeto da classe BTREE*

3 - APONTADOR INTERNO : a classe BTREE possui um apontador que aponta para o último dado que foi manipulado. Por exemplo, se houve um procedimento de busca de um certo dado, o dado pode ser ou não encontrado. Caso o dado seja encontrado, o apontador aponta para o dado, caso contrário, este fica apontado na posição onde o dado deveria estar. Se houve inserção, o apontador aponta para o dado inserido.

Este apontador aumenta a eficiência dos procedimentos de busca do PRÓXIMO elemento e do elemento ANTERIOR com relação ao elemento atualmente posicionado, assim como o posicionamento no PRIMEIRO e ÚLTIMO elemento da btree. Portanto, percorrer a btree do

início ao fim torna-se um procedimento trivial.

4 - PONTEIRO DE BTREE : a classe BTREE possui uma estrutura para guardar o número de dados armazenados na btree e o endereço, no arquivo, do início da btree. Esta estrutura é denominada PBTREE (ponteiro de btree). PBTREE representa bem um objeto da classe BTREE, pois o endereço, nela contida, informa a qual btree o objeto criado se refere. Assim, várias btrees podem ser armazenadas num mesmo arquivo sem nenhum problema e btrees podem, também, ser guardadas em estruturas de armazenamento como se fossem elementos atômicos, tal qual um "string" por exemplo. Este armazenamento se dá gravando PBTREE. Tal estrutura permite, portanto, representar elementos que sejam conjuntos de conjuntos.

Quando uma determinada btree deve ser manipulada, um objeto da classe BTREE é inicializado. Para que o objeto saiba sob qual btree ele tem controle, um ponteiro para PBTREE deve ser um dos parâmetros de inicialização. Portanto, ao final de uma manipulação com qualquer btree, a estrutura PBTREE tem que ser gravada.

5 - FLEXIBILIDADE NA FORMAÇÃO DA PRÓPRIA ESTRUTURA DA BTREE : há mais dois parâmetros que são considerados na inicialização de um objeto BTREE. Um é o número máximo de elementos contido numa folha base da btree e outro é o número máximo de índices contidos numa folha de índice da btree. Desta forma, o usuário da btree decide qual a forma mais adequada de dispor uma btree de acordo com a quantidade de dados que serão armazenados.

Caso estes parâmetros não sejam fornecidos, há valores por omissão que são nove (9) para o número máximo de elementos numa folha base e três (3) para o número máximo de índices numa folha índice.

3.5. EDIÇÃO

EDIÇÃO é o módulo responsável por criar MOSTRADORES para manipulação dos dados armazenados, ou seja, oferece um ambiente de edição para esboçar as janelas que servem de interface para dados serem manipulados por usuários.

Este ambiente consta de um EDITOR de texto/janela com funções básicas tais como apagar uma linha, acrescentar uma linha e posicionar no início da linha; e de um MENU barra contendo funções como escolher cores para edição, escolher o contorno da janela entre contornos duplo, simples ou misto, gravar em arquivo um esboço editado, abrir um novo ambiente para uma nova edição e carregar o conteúdo de um arquivo para o ambiente.

Para desenvolver tal ambiente fez-se uso de duas classes independentes : classe MENU e a classe EDITOR. A primeira preocupa-se em estruturar um menu com características internas próprias e funções externas para sua manipulação e a segunda preocupa-se em estruturar um editor com funções para sua edição.

Uma vez desenhada a janela , esta é gravada num arquivo de extensão .JAN. O arquivo tem características próprias contendo um cabeçalho inicial que designa a largura e comprimento da janela, assim como a posição (Linha,Coluna) onde deve ser posicionada a janela na tela.

Tendo este arquivo, há uma classe INTERF que é uma classe que decodifica o arquivo, apresenta a janela na tela e manipula com os atributos que estão ali dispostos.

Quando a janela está sendo desenhada, o caracter arroba (@) designa que na posição onde ele se encontra deve ficar posicionado um atributo do tipo que está especificado depois de @. Na decodificação, os atributos são devidamente arranjados ao longo da janela desenhada.

3.6. SUMÁRIO

Descreveu-se nos tópicos anteriores a finalidade de cada um dos módulos implementados neste trabalho. O módulo principal é o módulo onde definem-se os tipos. Os tipos constituem uma representação para os dados. Através deles busca-se uma maior equivalência entre o dado que deve ser representado e a forma de representação.

Os tipos definidos permitem uma representação bastante variada de dados tanto na forma como se apresentam quanto na sua complexidade. Isto porque, a partir dos tipos disponíveis é possível combiná- los e formar novos tipos para representar novos dados.

Suas definições tiveram base em observações de informações reais que necessitam ser representadas, como por exemplo informações de múltipla escolha (tipo Menu) e informações condicionais (tipo União), assim como no estudo do modo de representação de elementos da Teoria de Conjuntos, onde a partir de átomos é possível representar todos os outros elementos, combinando-os através de n_uplas e conjuntos.

Depois de oferecer uma forma de representação para dados, tratou-se em manipulá-los para que se pudesse validar a sua representação. Por simplificação, esta manipulação consiste na apresentação de um tipo na tela para que dados que satisfazem tal representação possam ser armazenados, removidos e lidos de arquivos. Para isto classes adequadas foram definidas.

Teve-se o cuidado de definir e implementar todas as classes utilizadas para montar o sistema, de modo a satisfazer propriedades de modularidade e extensão para serem utilizadas em outros contextos e permitirem uma fácil alteração.

CAPÍTULO 4

EXEMPLO DE APLICAÇÃO E ESTRUTURA DE "FRAMES"

4.1. GENERALIDADES

Utilizando a metodologia de orientação a objetos, o presente trabalho desenvolve tipos abstratos com o objetivo de representar informações , organizá-las ordenadamente e gravá-las em arquivo para acessos posteriores.

Neste capítulo é apresentado um exemplo de aplicação para ilustrar as ferramentas desenvolvidas. É mostrado como gerar uma representação a partir da combinação dos tipos disponíveis. Depois, há a fase de desenhar um esboço para esta representação, e finalmente a manipulação, que consiste na leitura, armazenamento e remoção de dados nesta representação definida, utilizando a janela desenhada.

O último tópico enumera as características de "frames" e discute os aspectos encontrados nos tipos abstratos desenvolvidos que permitem expressar com facilidade esta maneira de representar dados.

4.2. FICHA PESSOAL DE ALUNO

No exemplo utiliza-se uma ficha pessoal de aluno (empregada no departamento de atendimentos de alunos da Unicamp, veja apêndice B),com o objetivo de representá-la para que posteriormente alunos possam ser cadastrados.

Primeiramente, é feita uma análise para identificar os tipos que serão atribuídos a cada atributo contido na ficha.

Através da função campo_mensag() descreve-se uma combinação de tipos numa certa ordem, até obter a descrição de toda a ficha. Neste momento o **campo_mensageiro** da ficha é criado. Através de um objeto da classe CAMPO_MENS, este **campo_mensageiro** é manipulado, podendo ser, por exemplo, gravado num arquivo para que posteriormente possa ser utilizado.

A criação do **campo_mensageiro** para a FICHA PESSOAL DE ALUNO é feita da seguinte maneira (esta descrição pode ser acompanhada, paralelamente, observando o programa <ficha.cpp> no apêndice B) :

Inicia-se com um TIPO RAIZ , que como já foi dito, é um tipo que armazena o conteúdo do tipo que está ligado diretamente a ele, numa posição fixa do arquivo. Este TIPO RAIZ tem um TIPO BTREE ligado a ele, cuja chave é um TIPO LONGO que representa o atributo Número do Registro Acadêmico do Aluno e tem como dado um TIPO ENDEREÇO que está ligado a um TIPO PRODUTO VETORIAL que contém os tipos que representam os atributos restantes da ficha. O TIPO ENDEREÇO é considerado aqui, como precaução para o caso do conteúdo do TIPO PRODUTO VETORIAL não caber junto com os outros tipos já definidos numa mesma folha em arquivo, pois as estruturas de arquivo são formadas por folhas de no máximo de 512 caracteres.

No TIPO PRODUTO VETORIAL há tipos:

1 - MENU : que é para os atributos Nível (graduação / pós_graduação), Sexo (Masculino / Feminino) e País (Brasil / Exterior);

2 - UNIÃO : composto pelo TIPO PRODUTO VETORIAL e TIPO VAZIO e vinculado ao TIPO MENU que representa um atributo com as seguintes escolhas: TEM ENDEREÇO FORA DE CAMPINAS / NÃO TEM ENDEREÇO FORA DE CAMPINAS. Se a primeira opção for escolhida o tipo ativo na UNIÃO é o TIPO PRODUTO VETORIAL, que representa atributos para descrição do endereço fora de Campinas e se a segunda opção for a escolhida o tipo ativo na UNIÃO é o TIPO VAZIO, ou seja, não há atributos a considerar;

3 - INTEIRO : representa atributos de datas;

4 - STRING : representa atributos como nomes das cidades, dos pais e do aluno.

Neste exemplo nem todos os tipos são necessários para a sua representação. Não é utilizado o TIPO DOUBLE E TIPO LISTA.

A figura **FIG4.1** apresenta um esquema da ficha representada pelos tipos.

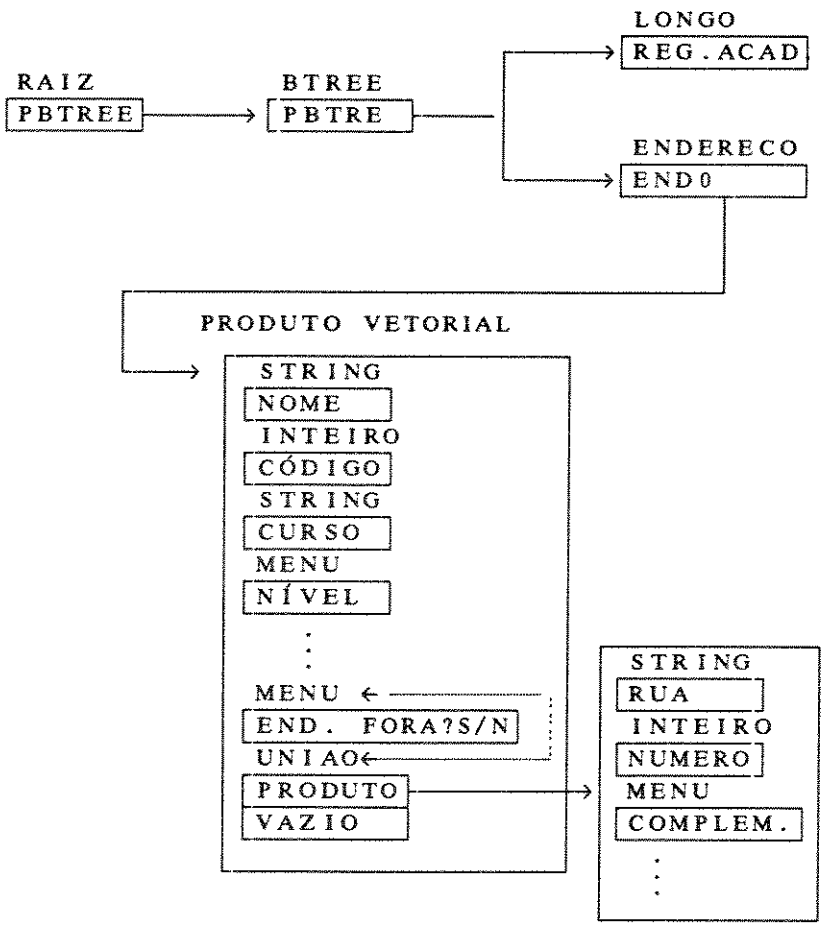


FIG4.1 - *Ficha Pessoal de Aluno representada por tipo*

Depois de descrito o tipo adequadamente, utilizando o editor de texto/janela , desenha-se a janela para o tipo descrito. As posições onde os atributos ficam na janela é indicado através da utilização do símbolo '@' seguido de um número. Por exemplo,

'@3' indica que na posição do caracter @ deve ficar posicionado o terceiro atributo descrito pela função **campo_mensag()**.

Este desenho da janela é feito através do editor, ativado pelo comando :

> ej

Dentro do ambiente, como mostra a figura **FIG4.2**, o menu é ativado pela tecla <Alt M> e faz-se uma escolha, posicionando sobre as opções do menu com as setas ESQ, DIR, P/CIMA, P/BAIXO ou digitando a letra que está em maiúscula na opção desejada.

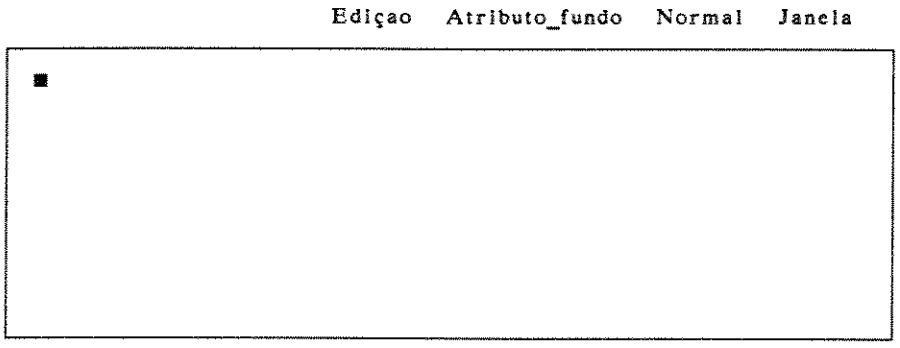


FIG4.2 - *Ambiente do editor de janelas*

onde :

Ediçao - menu que contém as opções relacionadas com a edição de arquivos de janela : Novo, Grava, Regrava, Configura e Abandona.

Atributo_fundo - apresenta as cores possíveis de fundo e quando é feita a escolha um menu Atributo_frente é apresentado tendo como cor de fundo a escolhida no menu anterior. Neste novo menu é possível escolher a combinação frente_fundo da cor para edição.

Normal - permite alternar entre atributo Piscante e Não_Piscante.

Janela - há quatro escolhas da forma de contorno da janela :
╔ , ┌ , ╓ , ╒ .

Finalmente, tendo o tipo definido gravado em arquivo e a janela que representa o tipo, também em arquivo, há um programa manipulador de tipos que é ativado pelo comando :

> mr <*arq* - nome do arquivo contendo uma representação por tipos>

Este programa manipulador segue os seguintes passos :

1 - É feita a leitura do **campo_mensageiro** contido no arquivo *arq*;

2 - Este **campo_mensageiro** é passado como parâmetro para se inicializar um objeto da classe CAMPO;

3 - O objeto CAMPO ativa a função **edita()**, que permite que dados sejam editados e também gravados ou removidos do arquivo de dados, até que uma tecla de interrupção que abandone o processo seja apertada;

4 - Para terminar a edição, o objeto executa a função **retorna_edita()**, que faz a limpeza adequada da tela e retorna ao sistema operacional.

Quando termina a operação, um arquivo de extensão .REM ou .FIC, que se refere a um arquivo manipulado por um objeto REMENDO ou um objeto FICHÁRIO , contém todos os dados referentes às operações feitas.

A função **edita()** controla as teclas apertadas e faz com que o atributo, no momento ativo, execute um procedimento relativo à tecla apertada, por exemplo a tecla <CRTL D> faz com que o atributo ativo execute o procedimento **deft()** (atribuir o valor 'default' ao atributo).

Observe que a função **deft()** é ativada sem saber qual tipo de atributo se trata. Neste caso, é só no momento da execução que há a correspondência entre objeto/procedimento e a ação devida é executada (característica de polimorfismo com ligação tardia).

Portanto, para representar uma informação basta compor adequadamente os tipos com suas respectivas características e restrições.

Porém, no estágio em que se encontra o trabalho, a cada novo tipo elaborado que se queira definir, cria-se um programa que deve ser compilado para que ao ser executado gere um **campo_mensageiro** que é gravado em arquivo.

Na continuação do trabalho, um dos objetivos é desenvolver um ambiente para definição de tipos, onde seja possível criar tipos combinados e que eles possam ser gravados convenientemente em arquivos. Este ambiente deve permitir acesso a estas representações para eventuais consultas ou alterações em suas características.

4.3. Estrutura de "FRAMES"

Minsky [MIN75], em 1975, apresentou um trabalho onde descrevia idéias, baseadas em pesquisas , de uma estrutura para representação do conhecimento, a qual denominou "FRAMES".

E lança a seguinte idéia :

" Um "frame" é uma estrutura de dados para representação de um situação estereotípica. Ligado a cada "frames" estão alguns tipos de informações. Seria uma rede de nós e relações. O topo de um "frame" é fixo e representam coisas que são sempre verdades a respeito da situação, os níveis inferiores têm muitos terminais (SLOTS) que devem ser preenchidos com dados específicos e também podem especificar condições que suas atribuições têm que encontrar."

Segundo Hayes [HAY77], o trabalho de Minsky trouxe uma idéia para uma estrutura de dados e esta idéia foi exposta de uma forma bastante ampla, não deixando de todo claro uma definição exata do que é "frame". Desta forma, diferentes versões e tentati-

vas de implementação de uma linguagem baseada em "frames" vem sendo desenvolvida, restringindo cada versão a alguns tipos específicos de área de aplicação.

No que se segue, é apresentado as características encontradas numa estrutura que é especificada segundo conceitos de representação por "frames".

Um "frame" é uma estrutura de representação genérica para um conjunto de objetos com atributos bem definidos. Estes atributos são designados, na literatura, por SLOTS.

Estes atributos podem ser instanciados da seguinte maneira :

1 - Instanciação por Valor : os atributos assumem valores de um certo tipo bem definido, por exemplo um valor inteiro dentro de um certo intervalo;

2 - Instanciação por Procedimento : os valores dos atributos são obtidos em decorrência da execução de um determinado procedimento;

3 - Instanciação por 'default' : no caso de omissão de valor num certo atributo, este atributo pode ser instanciado com um valor 'default'.

Os atributos, geralmente, podem possuir certas restrições internas, que devem ser satisfeitas para todas as suas instâncias. Por exemplo, um atributo do tipo inteiro que deve satisfazer a restrição de pertencer a um certo intervalo.

Pode haver procedimentos vinculados aos atributos que são ativados em decorrência de uma certa instância ou se houver a remoção de uma certa instância. Esta caracterização é discutida por Winston [WIN88], onde utilizando a linguagem LISP, desenvolve uma implementação usando um encadeamento de listas.

Entre "frames" deve haver uma abstração do tipo generalização/especialização. Isto permite que "frames" herdem valores de atributos de instâncias de frames_*pai*, e permite que "frames" se tornem mais especializados ao longo de novas especificações de acordo com a evolução da informação que está sendo representada. Esta especialização é a mesma considerada no mecanismo de herança entre classes.

Atributos de um "frame" podem ser do tipo "frame", definindo-se portando uma estrutura em cadeia. Isto é ilustrado no esquema da figura FIG4.3.

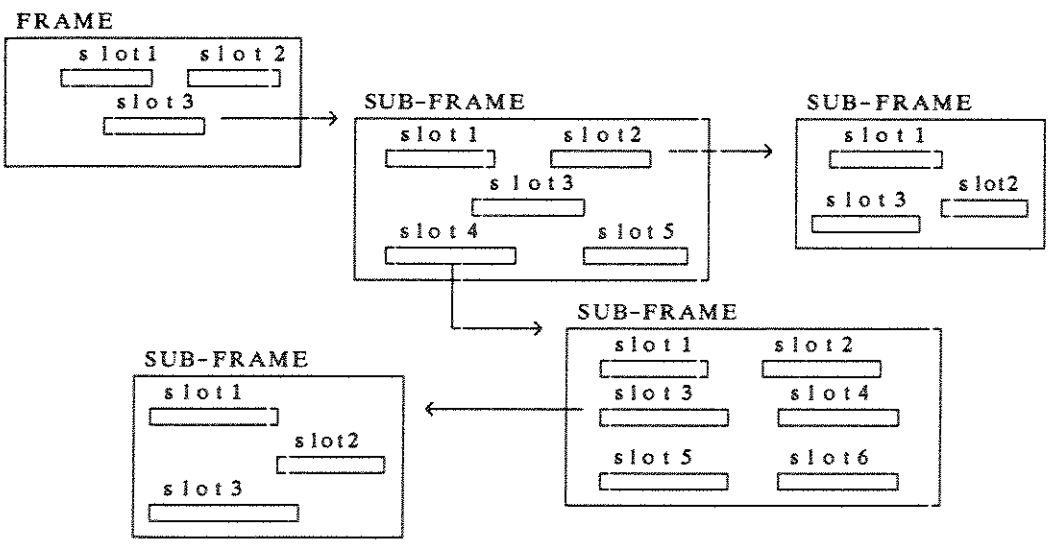


FIG4.3 - *Estrutura genérica de "frames"*

Levando-se em conta as características de "frames" anteriormente mencionadas, com os tipos abstratos implementados é possível criar estruturas de "frames" para representar informações. Os tipos abstratos disponíveis possuem condições para conter restrições e desencadear procedimentos de compatibilização em decorrência de instanciações. Há, também, uma função de verificação para avaliar se uma determinada instância está satisfazendo as restrições.

Entre os tipos definidos encontra-se o tipo Produto Vetorial que permite definir um "frame", que no caso é uma sequência de atributos de tipos variados, podendo, eventualmente, haver entre eles um tipo Produto Vetorial especificando um novo "frame".

A figura **FIG4.4** exibe um esquema de uma definição de um "frame" através dos tipos, onde um dos atributos é um "frame".

```
FR1 = campo_mensag(Produto,n,
                   campo_mensag(String,LARG_MAX,VALOR_DEFAULT),
                   campo_mensag(Btree,CHAVE,DADO),
                   campo_mensag(Produto,_,_,....),
                                  :
                                  :
                   campo_mensag(Inteiro,MAX,MIN,VALOR_DEFAULT));
```

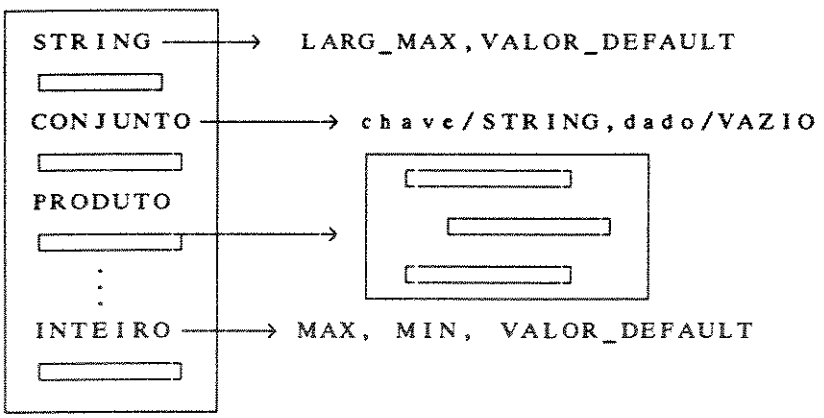


FIG4.4 - *Especificação de um "frame", onde um dos atributos é outro "frame"*

A notação para representação do "frame" FR1, adotada acima, está bastante simplificada para não sobrecarregar com especificações.

FR1 representa um "frame" constituído por alguns atributos, entre eles há um atributo tipo Produto Vetorial que representa um outro "frame".

Depois de definido um "frame", o seu apontador serve como uma variável para inicializar um objeto da classe CAMPO. Este objeto CAMPO tem disponível procedimentos para manipulação de um dado, sem fazer nenhuma restrição ou consideração dependendo do tipo de dado que está sendo manipulado.

Cabe reforçar aqui que a classe CAMPO explora bastante o polimorfismo, pois a decisão de qual procedimento realmente ativar é feita só durante a execução, em função dos tipos de atributos considerados.

Quanto ao aspecto de instanciação por procedimento, no estágio atual do trabalho, não é possível satisfazê-lo. Para isso é necessário implementar mais um tipo abstrato, que seria denominado tipo Função. Também, os aspectos de generalização/especialização não foram ainda considerados.

4.4. SUMÁRIO

Apresentou-se uma aplicação das ferramentas desenvolvidas para representação de informações. Mais especificamente, discutiu-se a representação por "frames", descrevendo-se suas principais características e enumerando, paralelamente, quais e como tais características podem ser modeladas de maneira natural utilizando a definição por tipos.

CAPÍTULO 5

CONCLUSÃO E SUGESTÕES DE TRABALHOS FUTUROS

5.1. ASPECTOS GERAIS DA METODOLOGIA DE ORIENTAÇÃO A OBJETOS

A metodologia de orientação a objetos tem a finalidade de projetar e implementar sistemas de software a partir da combinação de objetos. Estes objetos são instâncias de tipos abstratos (ou classes), adequadamente definidos a partir de técnicas utilizadas nesta metodologia.

Tais técnicas permitem uma maior abstração na programação, uma descentralização do sistema, um encapsulamento de implementações de dados, havendo interface através de seus serviços disponíveis. Isto leva , naturalmente, a uma fácil extensão de sistemas e uma reutilização de tipos abstratos em diferentes sistemas.

Porém, cabe ressaltar que não há um mecanismo que a partir de um sistema, seja possível definir, precisamente, os tipos abstratos com seus respectivos serviços que irão compor este sistema.

Tais definições se baseiam, atualmente, em experiências adquiridas ao longo do uso de tal metodologia. O trabalho permitiu a aquisição de uma experiência neste sentido : obter uma percepção no sentido de analisar um sistema a partir de objetos que o compõe.

O problema de definir tipos, de modo que seja possível combiná- los para representar dados compostos e que estes dados possam ser manipulados sem haver condicionais com relação ao tratamento, é resolvido de maneira natural, quando aplica-se a metodologia de orientação a objetos. Tal definição engloba especificação de características, restrições e procedimentos, que dados deste tipo definido devem ter, aumentando suas abstrações quando manipulados num sistema.

5.2. PROPOSTAS DE TRABALHOS FUTUROS

Como foi visto, os tipos especificados permitem a definição de estruturas de "frames". Através da função **campo_mens()**, uma estrutura interna (**campo_mensageiro**) é criada contendo todas as características do "frame" definido. Desta forma, a cada definição de um "frame", para representar uma dada informação, é necessário criar um programa que utiliza a função **campo_mens()**, e grava a descrição deste "frame" para manipulação posterior. Uma proposta para melhorar este desempenho, é desenvolver um ambiente onde "frames" possam ser definidos "on_line", ou seja, um editor de "frames" que permite uma entrada de informações que podem ser organizadas formando um **campo_mensageiro** do "frame" que se define.

Este editor de "frames" pode ser desenvolvido utilizando-se os próprios tipos para se inicializar objetos CAMPO que manipularão com a entrada de dados para especificação de "frames".

A ilustração **FIG5.1** apresenta a especificação de um "frame", onde cada atributo é especificado com suas características e restrições, sendo lidos via atributos de tipos pré_definidos. Por exemplo, na especificação do ATRIB3, há um atributo tipo String, que entra com o "string" "Inteiro", em decorrência disto três (3) atributos tipo Inteiro são acionados para que seja colocado os valores *máximo*, *mínimo* e *"default"* para

tal atributo.

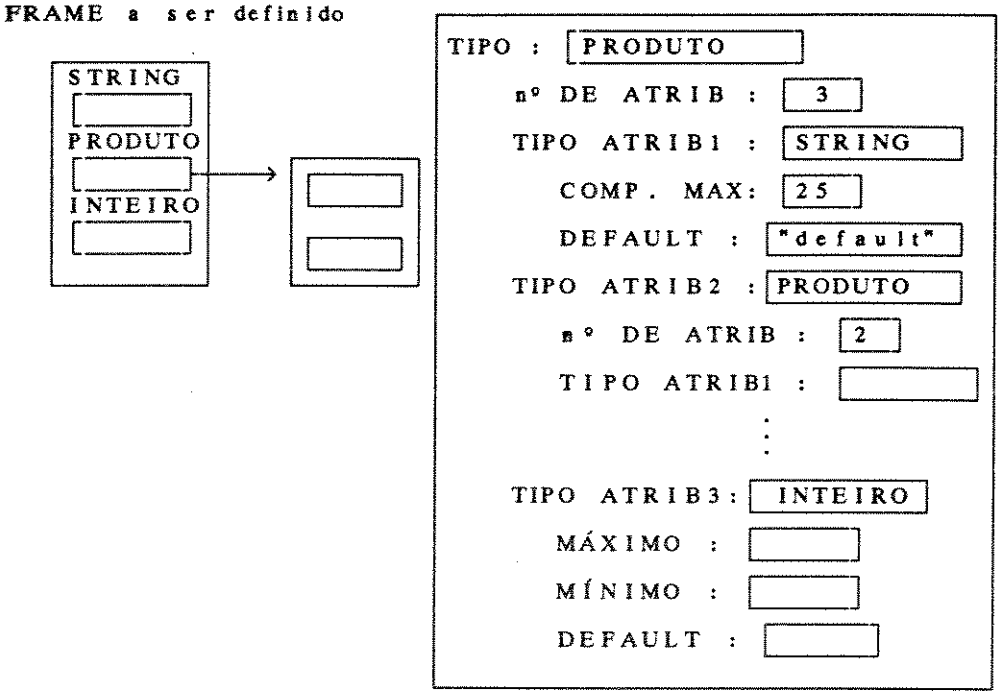


FIG5.1 - *Ilustração da utilização de um editor de "frames"*

Outro melhoramento a se considerar é definir um novo tipo, designado tipo Função, que representa conjuntos. Este tipo relaciona-se com outros tipos executando operações como soma ou multiplicação com seus conteúdos e guarda o resultado da operação como conteúdo próprio. É um tipo vinculado a outros tipos. O dado de um atributo deste tipo é sempre obtido pela execução de uma operação. Não é um *tipo por valor*, mas um *tipo por função*, ou seja, o usuário não tem controle sobre o dado, mas o dado é obtido internamente dependendo dos valores contidos nos atributos vinculados a este e da função contida no atributo que foi inicializado como tipo Função.

A princípio, seria um tipo que deveria conter um ponteiro para um função já compilada, que seria ativada sempre que houvesse uma instanciação. Mas, pensando num tratamento mais cuidadoso, poderia se pensar numa linguagem onde os procedimentos pudessem ser definidos. Haveria uma perda no tempo de execução, que não seria de todo negativo, devido ao ganho no aumento de abstração, já que neste caso o próprio gerador dos tipos definiria suas funções e não estariam pré_estabelecidas pelo implementador dos tipos.

Tal tipo especificaria atributos de "frames" que são instanciados através de funções.

A terceira consideração imediata é estabelecer Mecanismo de Herança entre os "frames", que vão sendo definidos, de modo que possa haver uma especialização de "frames" e também, uma herança de características de um "frame" para outro sem haver necessidade de repetições. No caso, uma nova classe deveria ser criada contendo procedimentos que permitissem acessos aos dados que lhe são disponíveis por herança num

tipo já definido e acessado através de um objeto da classe CAMPO.

Considerando a linguagem utilizada (C++ Zortech - versão 1.02 para ambiente MS_DOS) para desenvolver o sistema que foi proposto, houve limitações devido aos aspectos experimentais a nível de mercado que a versão de tal linguagem se destinou. Não havia possibilidade, por exemplo, de estabelecer heranças múltiplas entre classes definidas, e havia problemas no compilador, de maneira que algumas ferramentas da linguagem não se processavam adequadamente com relação ao fim ao qual se destinaram. Com isso, houve necessidade de adaptações, que inevitavelmente levaram a uma implementação não otimizada nos aspectos que se referem a explorar todas as ferramentas que se destinam à implementação de um sistema orientado a objeto.

É necessário reconsiderar certos desenvolvimentos quando continuar a presente implementação numa nova versão da linguagem.

Todo o sistema foi implementado em C++, não recorrendo a nenhuma ferramenta. Ao todo foram 9587 linhas de código fonte e 114Kbytes do sistema compilado.

5.3. Considerações finais

As classes, definidas para compor o sistema desenvolvido, poderão ser utilizadas isoladamente em aplicações distintas das propostas neste trabalho que se propôs. No que se refere à definição dos tipos as classes CAMPO, cada *classe* CAMPO_*tipo* derivada de CAMPO e CAMPO_MENS devem ser utilizadas conjuntamente, uma vez que elas estabelecem um relacionamento para que dados de um certo tipo possam ser manipulados.

Para efeitos de rapidez de implementação, devido ao tempo limitado, considerou-se a edição de tipos como um serviço disponível da classe CAMPO. Tal consideração não deve ser feita, pois a definição de um tipo não implica necessariamente de uma apresentação em tela. Na continuação do trabalho, tal serviço deverá ser isolado da classe CAMPO, e redefinido adequadamente numa outra classe que terá o papel de estabelecer uma apresentação na tela de dados de um certo tipo.

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO83] Aho, Alfred V. et al.. :
Data Structures and Algoritms
Addison_Wesley,1983

[AKA87] Akama, Seiki :
Presupposition and Frame Problem in Knowledge Bases
The Frame Problem in Artificial Intelligence, edited by Frank M Brown
pp 193-203, Proceedings of 1987 Workshop

[BAN88] Bancilhon, François :
Object-Oriented Database Systems
7th ACM SIGART-SIGMOD-SIGACT
Symposium on Principles of DataBase Systems, pp 1 -11, March 1988

[BAR90] Barbosa, Airton L. et al.. :
Considerações sobre o Signo : um sistema gerenciador de Objetos.
Anais do 5º Simpósio Brasileiro de Banco de Dados,pp 87 - 97, 1990

[BER88] Berry, John :
The Waite Group's C++ Programming.
Howard W. Sams & Company,1988

[FON90] Fonseca, Décio & Motz,Regina :
Estudo Formal da Estrutura de um Modelo de Dados Orientado a Objetos
Anais do 5º Simpósio Brasileiro de Banco de Dados, pp 113 - 127, 1990

[HAY85] Hayes, Patrick J. :
The Logic of Frames
Readings in Knowledge Representation, edited by
Ronald J. Brachman & Hector J. Levesque
Morgan Kaufmann Publishers,Inc., pp 287- 295, 1985

[KOR86] Korth, Henry F. & Silberchatz,Abraham :
Database System Concepts
McGraw_Hill,1986

[LIE88] Lieberherr, K. & Holland,I. & Riel A. :
Object-Oriented Programming: An Objective Sense of Style
pp 323 - 334, OOPSLA'88 Proceedings

[McI76] McIlroy
Software Engineering Concepts and Techniques
eds. J. M. Buxton, P. Naur and B. Randell, pp 88-89,1976

[MEY88] Meyer, Bertrand :
Object-Oriented Software Construction.
Prentice Hall,1988

[MIN75] Minsky, Marvin :
A Framework for Representing Knowledge.
The Psychology of Computer Vision, edited by
Patrick H. Winston, Mc Graw_Hill Book Company, New York, pp 211 - 282, 1975

[REI85] Reiter, Raymond :
On Reasoning by Default
Readings in Knowledge Representation, edited by
Ronald J. Brachman & Hector J. Levesque
Morgan Kaufmann Publishers,Inc., pp 401 - 410, 1985

[RIC83] Rich, Elaine :
Inteligência Artificial
McGraw_Hill - 1983

[STR86] Stroustrup, Bjarne :
The C++ Programming Language.
Addison-Wesley,1986

[STR88] Stroustrup, Bjarne :
What is Object-Oriented Programming?
IEEE Software, pp 10-20, May 1988

[ULL82] Ullman, Jeffrey D. :
Principles of Database Systems
Computer Science,1982

[WIE88] Wiener, Richard S. & Pinson,Lewis J. :
An Introduction to Object-Oriented Programming and C++.
Addison-Wesley,1988

[WIN84] Winston, Patrick Henry & Horn, Berthold Klaus Paul :
LISP
Addison Wesley, 1988

[WIN85] Winograd, Terry :
Frame Representation and Declarative / Procedural Controversy
Readings in Knowledge Representation, edited by
Ronald J. Brachman & Hector J. Levesque
Morgan Kaufmann Publishers,Inc., pp 358 - 370, 1985

[XAV90] Xavier, Carlos Magno da Silva et al .. :
Modelagem Conceitual Orientada a Objetos para Aplicações não Convencionais em Banco de Dados.
Anais do 5º Simpósio Brasileiro de Banco de Dados, pp 98 - 112, 1990

APÊNDICE A

PROGRAMAÇÃO ORIENTADA POR OBJETO E C++

C++ é uma extensão da linguagem C, que oferece ferramentas que não só facilitam na implementação de programas orientado a objetos como também tornam os programas mais eficientes e legíveis, e com isso mais próximos do modo como os vemos no mundo real.

Este apêndice enumera as ferramentas dessa linguagem com exemplos que ilustram suas manipulações.

A.1. CLASSES

Classe é um tipo de estrutura que permite agrupar dados ("member") e funções ("member functions") que manipulam com estes dados. Os dados e as funções contidos na classe podem ser **privados** ou **públicos**.

Um dado é **privado** quando pode ser manipulado somente pelas funções contidas na própria classe e uma função **privada** pode somente ser utilizada pelas funções **públicas** da classe. Funções **públicas** são aquelas disponíveis para manipulação do tipo abstrato que está sendo definido.

Um dado **público** permite que funções externas à classe o manipulem.

Em C++ o tipo de estrutura **struct** é um tipo especial de classe, pois todos os seus dados, assim como as funções nela contida, são consideradas públicas.

No exemplo **EX1**, observa-se a definição de uma estrutura do tipo **class** e a implementação de procedimentos da classe classe1.

A estrutura divide-se em duas partes :

- antes da palavra reservada **public** e
- depois da palavra reservada **public**.

Na primeira parte encontram-se os membros da classe que são reservados. Na segunda parte encontra-se os membros da classe que são públicos, ou seja, podem ser manipulados sem restrições.

Na definição da classe classe1, observa-se uma função que foi descrita **inline**. Esta liberdade de implementação faz com que o compilador gere o código deste procedimento junto com o código do programa que o utiliza, não havendo portanto uma chamada de função. Esta característica aumenta a velocidade de execução do programa e é sugerido quando o código do procedimento **inline** é bem pequeno, pois, caso contrário a eficiência de espaço é sacrificada em função de uma eficiência na velocidade, já que a cada utilização deste procedimento seu código será repetido. Mas, o compilador C++ ignora esta declaração quando o código do procedimento é demasiado grande, encarando-o como um procedimento convencional e fazendo acesso a ele via uma chamada de função.

```
primeira
parte
                class classe1
                {
                int x;
                int y;
          public :
segunda
parte
                classe1() { x = y = 0;}//descricaoinline
                void guarda(int,int);
                void imprime(void);
                };


          void classe1::guarda(inta,int b)
                {
                x = a;
                y = b;
                }

          void classe1::imprime()
                {
                printf("\nx = %d e y= %d",x,y);
                }
```

- *EXEMPLO* **EX1** -

No exemplo **EX2**, temos um programa principal que declara um objeto c1 como sendo da classe classe1, e no que se segue, o objeto c1 passa a manipular com as funções que lhe são próprias.

Na primeira chamada da função imprime(), o valor que é impresso em ambas as variáveis é zero, isto porque o construtor (explicado com mais detalhe posteriormente) inicializa as variáveis x e y com zero.

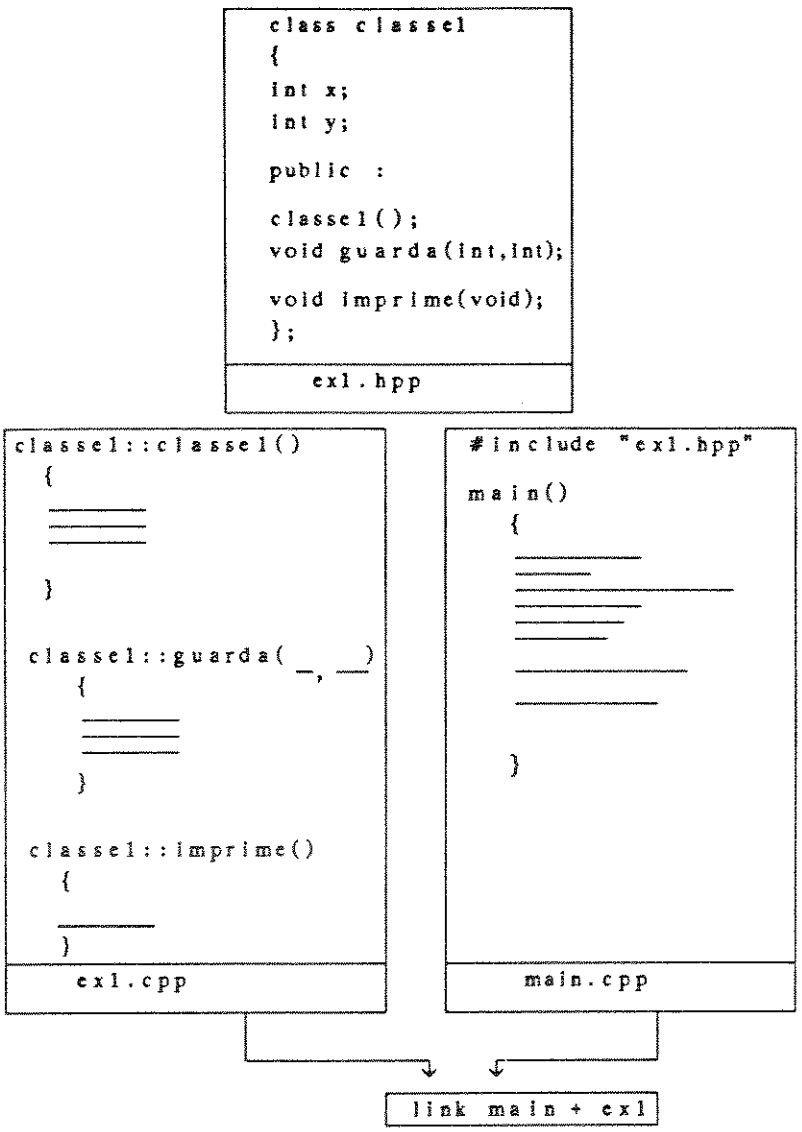
```
main()
   {
   classe1 c1;
   c1.imprime();   //imprimec1.x=0 e c1.y = 0
   c1.guarda(13,15)//atribuic1.x=13 e c1.y = 15
   c1.imprime();   //imprimec1.x=13 e c1.y = 15
   }
```

- *EXEMPLO* **EX2** -

Consideremos agora como é a distribuição destes programas com relação aos arquivos. A nível de exemplo podemos agrupar **EX1** e **EX2** em um único arquivo.

Em trabalho real a distribuição adequada é colocar a definição da classe em um arquivo de extensão .hpp (tornando-o um arquivo de cabeçalho - "header file") e as implementações das funções membro em outro arquivo com extensão .cpp.

Com o comando include do compilador e com o arquivo de extensão .cpp manipula-se com a clase criada de um modo modular.



- Distribuição adequada dos programas -

O papel do arquivo de extensão .hpp, quando declarado no comando include, é o de fornecer informações ao compilador da maneira como devem ser tratadas as funções que forem sendo encontradas no arquivo principal e que não estão declaradas no mesmo.

Observe que, separando-se o programa desta maneira, o arquivo, que contém a im-

plementação da classe declarada, funciona como um pacote à parte que poderá ser utilizado por um ou mais programas.

A.2. Declaração da função inline

Como foi visto no exemplo **EX1**, uma função foi declarada **inline**, e já se explicou qual o papel desta declaração. Cabe ressaltar aqui, que as funções ordinárias (funções que não são de nenhuma classe) também podem ter o mesmo tipo de comportamento.

O que difere é que as funções membros são consideradas **inline** implicitamente, bastando implementá-las no momento de sua declaração. Mas, com as funções ordinárias coloca-se a palavra reservada **inline** no início da implementação.

A.3. Construtores e Destruidores

No exemplo **EX3**, temos duas funções membros em destaque, que são o "**constructor**" e "**destructor**" da classe.

O "**constructor**" da classe é uma função que tem o mesmo nome da classe que está sendo descrita.

Ao se declarar um objeto como sendo da classe mensagem no exemplo **EX3**, automaticamente os procedimentos descritos no "**constructor**" são executados.

Quanto ao "**destructor**", é uma função membro de mesmo nome da classe descrita, com o símbolo ~ (til) na frente da função. O "**destructor**" é considerado complementar do "**constructor**". Esta função é executada, sem uma chamada explícita, sempre que se sai do escopo do objeto declarado.

Observe que com estas duas funções declaradas numa classe é possível executar procedimentos fora do corpo do programa principal, uma vez que haja uma declaração de um objeto como global. Neste caso, o escopo do objeto é todo o arquivo e portanto é possível executar um procedimento antes do main() (através do "**constructor**") e outro procedimento depois do main() (através do "**destructor**").

Pode-se destacar algumas inicializações típicas que ocorrem no "**constructor**":

- alocações internas de variáveis de ponteiros;
- atribuições de valores específicos às variáveis membros.

Tanto o "**constructor**" quanto o "**destructor**" não são obrigatórios na definição de uma classe.

```
class mensagem
    {
    char *s;
    int comp;
public:
    mensagem(char*);              // constructor
    ~mensagem() { delete s;}      // destructor inline
    char *le_buffer() { return s;} // inline
    void envia(char*);
    }
```

```
mensagem::mensagem(char* msg)
    {
    comp = strlen(msg);
    s = new char[comp+1];
    strcpy(s,msg);
    }
```

```
void mensagem::envia(char* p)
    {
    strcpy(s,p);
    }
```

- *EXEMPLO* EX3 -

A.4. Funções e Operadores overload

No exemplo **EX4**, a classe data possui duas funções membros com o mesmo nome. São as funções "overload". Dentro das declarações de classe, as funções podem ser "overload" sem nenhuma declaração explícita, o que não é o caso de funções ordinárias, pois nestas, inicialmente declara-se o nome da função que será "overload" e depois inicia-se suas implementações, como mostra o exemplo **EX6**.

Esta flexibilidade de se ter funções diferentes com o mesmo nome é adequado para se manter a coerência de procedimentos. Isto é, no exemplo **EX4** o que se quer com a função **data_nova()** é colocar uma nova data de dia, mês e ano nos membros privados, mas, o usuário pode entrar de duas maneiras diferentes com a data e portanto, tais datas devem receber tratamentos distintos, porém com o mesmo objetivo final. Isto é ilustrado no exemplo **EX5**.

Podemos também considerar "**constructor**" "overload", desta forma há maior flexibilidade, pois há procedimentos diferentes para inicialização da classe em questão.

```
class data
    {
    int dia,
        mes,
        ano;
public :
    data(int = 0,int = 0,int = 0);//inicialização com
                                  //valores 'default'
    void data_nova(int,int,int);
    void data_nova(char*);
    void fornece_data();
    };
```

- *DECLARAÇÃO DA CLASSE DATA DO EXEMPLO* EX4 -

```
data::data(int d, int m,int a)
      {
      data_nova(d,m,a);
      }

void  data::data_nova(int d,int m, int a)
      {
      if((d >= 1 && d <= 31) && (m >= 1 && M <= 12))
          {
          dia = d;
          mes = m;
          ano = a;
          }
     }

void  data::data_nova(char *dat)
     {
     char *dt = dat;
     dia = atoi(dt);
     while(*dt != '/') dt++; dt++;
     mes = atoi(dt);
     while(*dt != '/') dt++; dt ++;
     ano = atoi(dt);
     }

void  data::fornece_data()
      {
      cout << form("\nData : %d / %d / %d\n",dia,mes,ano);
      }
```

- *EXEMPLO* EX4 -

```
main()
   {
   data hoje;
   data ontem;
   hoje.data_nova(25,5,89);
   ontem.data_nova("24/05/89");
   hoje.fornece_data();
   ontem.fornece_data();
   }
```

- *EXEMPLO* EX5 -

```
overload f;
f(int x,int y)
  {
  ________
  ________
  ________________

   .
   .
   .
  }
f(char a,int s,char *c)
  {
  _________
  ________________
  _________

  .
  .
  .
  }
```

- EXEMPLO **EX6** *- função ordinária overload -*

Muitas vezes vê-se necessário redefinir operadores usuais da linguagem que estamos manipulando. Isto acontece, por exemplo, quando definimos uma classe dos números complexos e gostaríamos de somar ou multiplicar os números complexos. A linguagem C++ permite esta flexibilidade : redefinir todos os seus operadores (+ , *, /, -, &, !!, etc), porém a prioridade inicial se mantém.

Assim como nas funções membros, os operadores também podem se tornar "overload" sem uma declaração explícita, pois já o são considerados pelo compilador. Um exemplo que ilustra a sintaxe dos operadores "overload" encontra-se no exemplo **EX7** no apêndice C.

A.5. FUNÇÕES DO TIPO friend

Observe no exemplo **EX8**, que uma função é declarada **friend**. Esta declaração faz com que uma função ordinária tenha o mesmo privilégio que uma função membro de uma classe, que é ter acesso aos membros privados da classe.

Uma função ordinária declarada **friend**, apesar de passar a ter este privilégio, não tem as mesmas características de uma função membro, ou seja :

1 - A função, não sendo membro da classe, não está no seu escopo, logo o objeto da classe tem que ser um parâmetro da função para que ela possa acessar seus membros privados;

2 - A notação do ponto (.) não é adotada como nas funções membros;

3 - Sua implementação é feita de maneira ordinária, já que a função assim o é.

A declaração de uma função do tipo **friend** pode ser feita tanto na parte privada quanto na parte pública de uma declaração de classe.

Esta ferramenta aumenta a flexibilidade do programa, porém deve ser utilizada com cuidado para que apenas um número restrito de funções ordinárias tenham acesso aos dados privados, não violando assim a idéia de encapsulamento.

É possível, também declarar uma função membro de outra classe ou toda uma classe como **friend** da classe em questão.

No exemplo **EX8**, uma função ordinária é declarada **friend** em duas classes distintas, criando-se uma ponte entre as duas classes. Observe que a função carrega duas variáveis, que são objetos de cada uma das classes, com o objetivo de acessar seus dados privados.

```
class horario        //horario deve ser reconhecido como um tipo
class data
      {
int dia,
mes,
ano;
public:
            data(int d,int m,int a) { dia = d;mes = m;ano = a;}
            friend void data_horario(data,horario);
      };

class horario
      {
      long segs;
      friend void data_horario(data,horario);
public:
      horario(char*);
      };

horario::horario(char *t)
      {
      char *hr,*mn;
      hr = strtok(t,":");
      mn = strtok(0,":");
      segs = atol(hr)*3600+atol(mn)*60
      }

// definição da função friend

void data_horario(data d, horario h)
      {
      int hora = h.segs/3600;
      int min = (h.seg%3600)/60;
      printf("\n Data = %d / %d / %d e Horario = %d : %d \n",
d.dia,d.mes,d.ano,hora,min);
      }
```

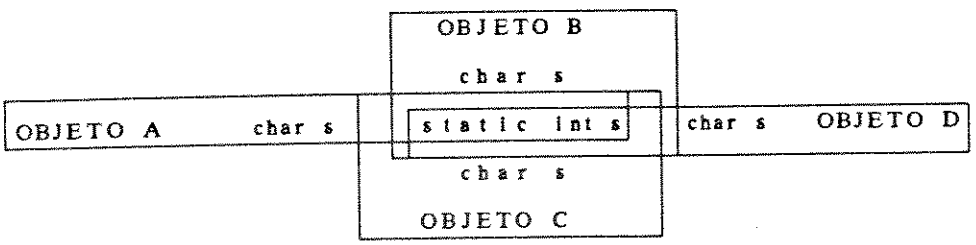
- Exemplo **EX8** -

A.6. VARIÁVEIS static E PONTEIRO DE REFERÊNCIA PARA CLASSE

Variável static : Uma vez definida uma classe, todos os objetos desta classe possuem seus próprios membros, mas talvez seja necessário que ao se declarar objetos de uma classe, eles tenham membros em comum e isto é possível declarando-se o membro da classe como **static**, veja o diagrama **D1**.

```
class ABCD
      {
      static int x;
char s;ABCD D;
      ...
      }
```

```
ABCD A;
ABCD B;//Inicialização de
ABCD C;// 4 objetos ABCD.
```

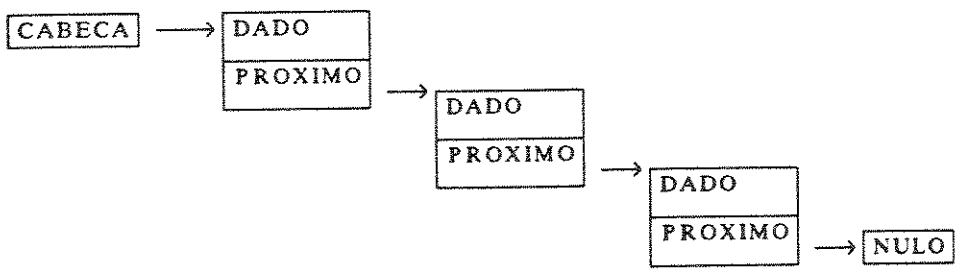


- *DIAGRAMA* **D1** -

Ponteiro de referência para classe : Um objeto tem associado a ele um ponteiro de auto_referência, ou seja, possui um ponteiro para o próprio objeto que está sendo manipulado. Este ponteiro está contido numa variável especial denominada **this**.

O exemplo **EX9**, que se encontra no apêndice C, ilustra as duas "ferramentas" acima. Refere-se à implementação de uma lista encadeada. Cada elemento da lista é um objeto da classe nó que possui um espaço para armazenar o dado e outro espaço para um ponteiro para o próximo nó da lista. Estes nós da lista possuem um cabeçalho comum de inicialização da lista (nó cabeça). Veja o diagrama **D2**.

Para implementar esta idéia, o conceito do ponteiro **this** e da variável **static** são utilizados, o primeiro para atribuição do próximo ponteiro para nó e o segundo para designar o nó cabeça.

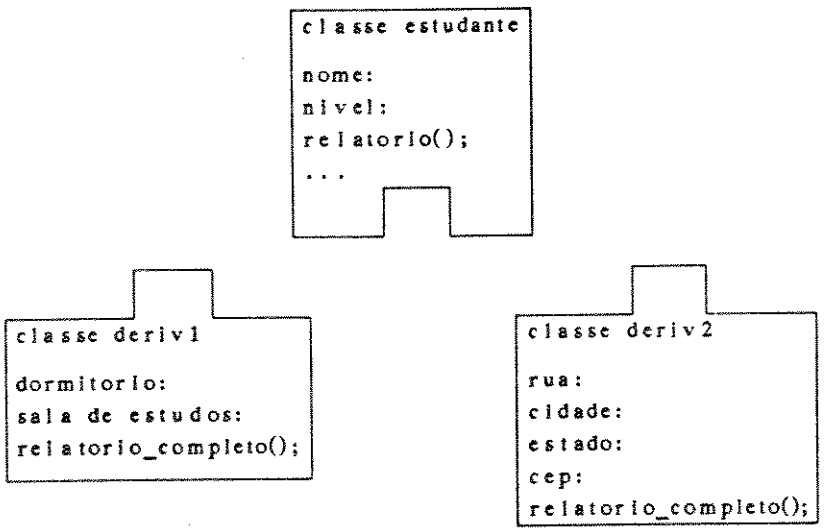


- *DIAGRAMA* **D2** -

A.7 - CLASSE DERIVADA

Considere a seguinte situação. Cadastramento dos alunos em uma universidade constando nome do aluno, nível escolar e endereço. Considere também, que há alunos que moram no campus e outros que moram fora do campus. Há características comuns entre os fichários e outras que se diferem. A melhor maneira seria colocar os itens comuns em uma ficha base e os itens divergentes em fichas distintas.

C++ oferece uma ferramenta que permite este tipo de organização. Com isto, há minimização de espaço, tempo de compilação, legilibilidade e modularidade. O diagrama D3 esquematiza esta ilustração.



- *DIAGRAMA* D3 -

Nos exemplos **EX10,EX10.1,EX10.2** no apêndice C,estão as implementações desta ilustração.

Existem certas características de classes derivadas que devem ser consideradas:

1- O compilador C++, inicialmente utilizado para implementação, permite que uma classe seja derivada de uma única classe base, mas a nova versão adquirida no final do trabalho já permite a derivação de classe a partir de mais de uma classe base;

2- As classes derivadas não possuem nenhuma diferença, a nível de compilador,de classes não derivadas. Desta forma, classes derivadas podem funcionar como classes bases para que novas classes possam ser derivadas destas;

3- Não há nenhum limite lógico quanto ao comprimento da cadeia de classes derivadas.

A.8. FUNÇÕES virtual

Uma outra ferramenta que permite ter um procedimento 'default' na classe base é o conceito de função **virtual**.

Com uma função **virtual** definida na classe base, pode-se ter a mesma função redefinida ou não na classe derivada. Se a função não for redefinida na classe derivada, um objeto desta classe utiliza a função da classe base. Porém, se a função for redefinida numa classe derivada o compilador substitui a definição original pela nova, sempre que a função é utilizada por um objeto desta classe derivada. O ponteiro para esta função é considerado como um membro da classe e é carregado para a pilha no momento da declaração de um objeto.

Com este tipo de declaração de função, torna-se possível explorar o polimorfismo com ligação tardia ou adiantada.

No exemplo **EX11**, no apêndice C, há a implementação de um programa utilizando função **virtual**.

A.9. VARIÁVEIS DO TIPO REFERÊNCIA

Além das declarações de variáveis usuais de C, C++ tem um terceiro tipo de declaração : o tipo referência. Como se sabe, uma variável regular ocupa um espaço para um dado e uma variável do tipo ponteiro ocupa um espaço que contém o endereço para um dado que será colocado em outro lugar. Quanto à referência, ela é um tipo que tem características tanto de uma variável regular como de ponteiro. A referência é um ponteiro para um dado, portanto não reserva espaço para conter um dado ao qual se refere, mas quando manipulamos com uma variável deste tipo, é como se manipulássemos com uma variável regular. Isto é uma forma de economia de espaço e facilidade de acesso.

Uma observação a destacar, é que uma variável do tipo referência tem sempre que ser inicializada no momento de sua declaração para que elas já possam apontar para um dado bem definido., o que não acontece com uma variável do tipo ponteiro. Isto porque, não é possível manipular com uma referência a nível de ponteiro, portanto há uma perda de manipulação, a nível de ponteiro, com um ganho de simplificação de notação, já que não há necessidade de utilizar a notação do asterisco (*).

Este tipo de variável é muito útil, por exemplo, quando se quer utilizar uma variável em uma função que modifica o seu conteúdo, ou utilizar como variável de uma função qualquer, para economizar espaço da pilha, pois desta forma o dado em questão não é colocado na pilha, mas apenas o ponteiro para o mesmo.

O exemplo **EX12** implementa um programa utilizando uma variável do tipo referência.

```
struct est
        {
        int x;
        int y;
        est(int,int);
        };

est::est(int a,int b) { x = a; y = b;}

funcao(est& R) { R.x = 13; R.y = 15; }

main()
        {
        est a(2,3);
        funcao("a.x = %d e a.y = %d",a.x,a.y);
              // sairá impresso a.x = 13 e a.y = 15
```

- *Exemplo* **EX12** -

A.10. SUMÁRIO

Este apêndice descreveu as características de C++, para se implementar um projeto orientado a objeto, apresentando alguns exemplos. Stroustrup [STR88] e Wiener [WIE88] descrevem detalhadamente tais características.

APÊNDICE B

FICHA PESSOAL DE ALUNO

```
/* FICHA PESSOAL DE ALUNO                    (09/08/90)  (ficha1.cpp) */

#include "ncampo.hpp"

char *sexo[2] = {"masc","fem "};
char *pais[2] = {"Brasil","Outro "};
char *nivel[2] = {"graduacao","pos_grad "};
char *codigo[8] = {     "FUNCAMP                ",
                        "UNICAMP                ",
                        "Convenio               ",
                        "Cortesia Diplomatica   ",
                        "Transferencia Regular  ",
                        "Tranferencia por Lei   ",
                        "Transferencia Convenio ",
                        "Complementacao         "};

char *outro_end[1] = {"End. fora de Campinas :"};

char *completa[2] = {"Num :","Dest:"};

main()
        {
        Campo_Menu *CMP = Campo_Mensag(Produto,"ficha_pes",0,0,0,37,
                Campo_Mensay(Inteiro,31,1,0,0,3),
                Campo_Mensag(Inteiro,12,1,0,0,11),
                Campo_Mensay(Inteiro,99,0,0,0,63),
                Campo_Mensag(Inteiro,12,1,0,0,11),
                Campo_Mensay(Inteiro,99,0,0,0,86),
                Campo_Mensag(Inteiro,12,1,0,0,1),
                Campo_Mensay(Inteiro,99,0,0,0,83),
                Campo_Mensag(String,40,0,3,0,0,1,"&Nome do Aluno"),
                Campo_Mensay(Menu,2,sexo,0,7,7,2,72,0,0,1),
 /*1*/  Campo_Mensag(String,11,0,3,0,0,1,"RG"),
                Campo_Mensay(String,40,0,3,0,0,1,"&Nome do Pai"),
                Campo_Mensag(String,40,0,3,0,0,1,"&Nome da Mae"),
                Campo_Mensay(Menu,2,pais,0,7,7,2,70,0,0,0),
                Campo_Mensag(String,20,0,3,0,0,1,"&Cidade Nata)"),
                Campo_Mensay(String,2,0,3,0,0,1,"UF"),
                Campo_Mensag(String,15,0,3,0,0,1,"Brasileiro(a)"),
                Campo_Mensay(Inteiro,99,0,0,0,11),
                Campo_Mensag(String,33,0,3,0,0,1,""),
                Campo_Mensay(Menu,2,nivel,0,7,7,2,60,0,0,0),
 /*2*/  Campo_Mensag(String,20,0,3,0,0,1,""),
                Campo_Mensay(Inteiro,9999,0,0,0,1230),
                Campo_Mensag(Menu,2,pais,0,7,7,2,70,0,0,0),
                Campo_Mensay(Longo,99999L,0L,0,0,38400L),
                Campo_Mensag(String,2,0,3,0,0,1,""),
                Campo_Mensay(Longo,99999L,0L,0,0,38400L),
                Campo_Mensag(String,2,0,3,0,0,1,""),
                Campo_Mensay(Menu,8,codigo,1,7,7,2,51,0,0,1),
                Campo_Mensag(String,40,0,3,0,0,1,""),
                Campo_Mensay(Inteiro,9999,1,0,0,1233),
 /*3&4*/  Campo_Mensag(Menu,2,completa,0,7,7,2,72,0,0,0),
                Campo_Mensay(Data1ao,"~2/29",0,0,2,
                        Campo_Mensag(Inteiro,9999,0,0,0,92),
                        Campo_Mensay(String,8,0,3,0,0,1,"Fundo")),
```

```
        campo_mensag(Longo,9999999L,0L,0,0,2343858L),
        campo_mensag(String,4,0,3,0,0,1,"034"),
        campo_mensag(String,28,0,3,0,0,1,""),
        campo_mensag(Menu,1,outro_end,0,7,7,2,51,0,0,-1),
        campo_mensag(Uniao,"-2/34",0,0,1,
                campo_mensag(Produto,"end",2,0,0,7,
                        campo_mensag(String,40,0,3,0,0,1,""),
                        campo_mensag(Inteiro,9999,1,0,0,1233),
                        campo_mensag(Menu,2,completa,0,7,/,2,72,0,0,0),
                        campo_mensag(Uniao,"-2/2",0,0,2,
                                campo_mensag(Inteiro,9999,0,0,0,92),
                                campo_mensag(String,8,0,3,0,0,1,"Fundo")),
                        campo_mensag(Longo,9999999L,0L,0,0,2343858L),
                        campo_mensag(String,4,0,3,0,0,1,"034"),
                        campo_mensag(String,20,0,3,0,0,1,""))),
                campo_mensag(String,2,0,3,0,0,1,""));

campo_mens *CHK = campo_mensag(Raiz,"fichal",7,2,"Ficha Pessoal de Aluno",
        campo_mensag(Btree,0,0,0,9,3,"","ch_fich",""",
                campo_mensag(Longo,999999L,/00000L,0,1,3943521),
                campo_mensag(Endereco,"",0,1,0,CHK")));
CHK->grava_mens("fichal");
delete CHK;
}
```

U N I C A M P

FICHA PESSOAL DE ALUNO

DIRETORIA ACADÊMICA

NÚMERO | NOME

CURSO — CÓD. | NOME

Nº NÍVEL — 1 - GRADUAÇÃO / 2 - PÓS-GRADUAÇÃO

DATA NASCIMENTO — DIA | MÊS | ANO

PAÍS — 0 - BRASIL / 1 - EXTER.

NATURALIDADE — C. E. P. | CIDADE | U. F.

NACIONALIDADE

SEXO — M MASC / F FEM.

CÉDULA DE IDENTIDADE — R. G. NÚMERO | U. F.

FILIAÇÃO

PAI:

MÃE:

SITUAÇÃO ESCOLAR — CÓDIGO | COLÉGIO OU FACULDADE REFERENTE À ÚLTIMA CONCLUSÃO

PAÍS — 0 - BRASIL / 1 - EXTER.

CONCLUSÃO EM: — CIDADE (C.E.P.) | U.F. | DIA | MÊS | ANO

CONCURSO VESTIBULAR — CIDADE (C.E.P.) | U.F. | MÊS | ANO

CÓDIGO

F - FUVEST
U - UNICAMP
D - CONVÊNIO
I - CORTESIA DIPLOMÁTICA
T - TRANSFERÊNCIA REGULAR
L - TRANSFERÊNCIA POR LEI
V - TRANSFERÊNCIA CONVÊNIO
C - COMPLEMENTAÇÃO

ENDEREÇOS:

EM CAMPINAS — RUA, AVENIDA, PRAÇA, ETC. | NÚMERO | COMPLEMENTO; BAIRRO | TELEFONE | DDD

FORA DE CAMPINAS — RUA, AVENIDA, PRAÇA, ETC. | NÚMERO | COMPLEMENTO; PAÍS | C. E. P. | CIDADE | UF | BAIRRO | TELEFONE | DDD

UNICAMP, / /

ASSINATURA DO ALUNO

APÊNDICE C

EXEMPLOS QUE ILUSTRAM ALGUMAS FERRAMENTAS DE C++

```
//---------------------- Exemplo EX7 ------------------------------
// ----------------- Operadores OVERLOAD --------------------
#include (stdio.h)

class complexo
        {
        double real;
        double imag;
public:
        complexo()(real = imag = 0;)
        complexo(double r,double i) (real = r; imag = i;)
        complexo operator-();   //oposto
        friend complexo operator+(complexo&,complexo&);
        friend complexo operator+(complexo&,double&);
        complexo operator-(complexo&);
        void imprime(void);
        };

complexo complexo::operator-()
        {
        real = -real;
        imag = -imag;
        return(*this);
        }

complexo complexo::operator-(complexo& z1)
        {
        complexo z;
        z.real = real - z1.real;
        z.imag = imag - z1.imag;
        return z;
        }

void complexo::imprime()
        {
        if(imag ( 0) printf("\n Z = %lf - %lfi\n",real,-imag);
        if(imag == 0) printf("\nZ = %lf\n",real);
        else printf("\nZ = %lf + %lfi\n",real,imag);
        }

//---------- Funcoes ordinarias friend da classe ---------------
complexo operator+(complexo &z1,complexo &z2)
        {
        complexo z;
        z.real = z1.real+z2.real;
        z.imag = z1.imag+z2.imag;
        return z;
        }

complexo operator+(complexo &z1,double &d)
        {
        complexo z;
        z.real = z1.real + d;
        z.imag = z1.imag;
        return z;
```

```
        }

//---------------- Programa Principal --------------------------------
main()
{
double d = 12.0;
complexo z,z1(-2,3),z2(1,0),z3(8,-7);
z = (z1+z2)+(-z3+d)-(z1-z3);
z.imprime();
}
```

```
//--------------------- Exemplo EX9 -----------------------------
//---------- Variavel static e ponteiro de referencia para classe ---------


#include (string.h)

class no
        {
        static no *cabeca;
        no *proximo;
        char *dado;
public:
        no(char* = 0);
        void imprime();
        };

no::no(char *ptr)
        {
        if(ptr != 0)
                {
                dado = new char[strlen(ptr)+1];
                strcpy(dado,ptr);
                proximo = 0;
                no *cursor = cabeca;
                while(cursor->proximo != 0) cursor = cursor->proximo;
                cursor->proximo = this;
                }
        else
                {
                dado = new char[strlen("raiz")+1];
                strcpy(dado,"raiz");
                proximo = 0;
                cabeca = this;
                }
        }

void no::imprime()
        {
        no *cursor = cabeca->proximo;
        while(cursor != 0)
                {
                printf("%s\n",cursor->dado);
                cursor = cursor->proximo;
                }
        }

main()
{
no n;
no n1("TESTE");
no n2("relatorio");
no n3("exposicao");
n.imprime();
}
```

```
//------------------ Declaracao de uma Classe Base

class estudante
        {
        char* nome;
        char nivel;     //grau do estudante:G=graduacao,M=mestrado,D=doutorado;
public:
        void relate();
        void cria_dado(char*,char
        };
```

```
#include "base.hpp"
#include (stream.hpp)
#include (string.h)

//---------- Implementacao da Classe Base ----------------------
//------------   EX10 ------------------------------------------

void estudante::relata()
        {
        cout (( form("\nNome: %s\nNivel = %c\n",nome,nivel);
        }

void estudante::cria_dado(char* n,char nl)
        {
        nome = new char[strlen(n)+1];
        strcpy(nome,n);
        nivel = nl;
        }
```

```
//------------------ Exemplo EX10.1 --------------------------
//---------------- Classe Derivada -------------------------

#include "base.hpp"
#include (stream.hpp)
#include (string.h)


//--------------------- Exemplo de classe derivada   -------------------------


//------------- Declaracao de uma Classe Derivada ----------------------- --


class interno_campus : public estudante
        {
        char *dormitorio,
             *sala_estudos;
public:
        void relata_completo();
        void novoestudante(char*,char,char*,char*);
        };



//------------- Implementacao de uma Classe Derivada da classe estudante------

void interno_campus::relata_completo()
        {
        relata();
        cout((form("Dormitorio: %s\nSala: %s\n",dormitorio,sala_estudos);
        )

void interno_campus::novoestudante(char* n,char nl,char* d,char* s)
        {
        cria_dado(n,nl);
        dormitorio= new char[strlen(d)+1];
        strcpy(dormitorio,d);
        sala_estudos = new char[strlen(s)+1];
        strcpy(sala_estudos,s);
        )

//------------ Programa Principal-----------------------------

main()
{
estudante x;
x.cria_dado("Joao Carlos",'G');
x relata();

interno_campus y;
y.novoestudante("Paula Fernanda",'M',"Rua 23","M1204");
y.relata_completo();
}
```

```
//------------------------- Exemplo EX10.2 ----------------------------
//---------------------- Classe Derivada ------------------------

#include "base.hpp"
#include (string.h)
#include (stream.hpp)

//---------------- Exemplo da Classe derivada2 -----------------------------

//------------ Declaracao da classe derivada2 ------------------------

class externo_campus : public estudante
        {
        char *rua,
             *cidade,
             *estado,
             *cep;
public:
        void relata_completo();
        void novoestudante(char*,char,char*,char*,char*,char*);
        );

//------------ Implementacao da classe derivada2 da classe estudante ----

void externo_campus::relata_completo()
        {
        relata();
        cout((form("End = %s\n%s,%s CEP = %s\n",rua,cidade,estado,cep);
        )

void externo_campus::novoestudante(char* n,char nl,char* r,char* cid,char* est,char *c)
        {
        cria_dado(n,nl);
        rua = new char[strlen(r)+1];
        strcpy(rua,r);
        cidade = new char[strlen(cid)+1];
        strcpy(cidade,cid);
        estado = new char[strlen(est)+1];
        strcpy(estado,est);
        cep = new char[strlen(c)+1];
        strcpy(cep,c);
        )

//------------------- Programa Principal ---------------------------------

main()
{
estudante x;
x.cria_dado("Maria Cecilia",'6');
x.relata();

externo_campus z;
z.novoestudante("Paulo Otavio",'D',"Av. Moraes Sales,425","Campinas","SP","13015");
z.relata_completo();
}
```