Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

AUTOMAÇÃO DO PROCESSO DE PROJETO E IMPLEMENTAÇÃO DE CLASSES EM AMBIENTES ORIENTADOS POR OBJETOS

por

Mark Alan Junho Song
Roberto da Silva Bigonha

RT 023/96

Caixa Postal, 702

30.161 - Belo Horizonte - MG

junho de 1996

Contents

Resumo

Como a organização de um programa é baseada nos tipos que este manipula e o estilo da programação orientada por objetos encoraja o reúso de código, o conhecimento e recuperação de tipos existentes na biblioteca são aspectos importantes no projeto e implementação das novas classes. A proposta deste trabalho é projetar e implementar uma ferramenta que automatize parte do processo de projeto e implementação de classes auxiliando na recuperação de componentes das bibliotecas.

Abstract

As program organization is based on types it manipulates and the style of object-oriented programming encourages code reuse in the development of programs, the knowledge and retrieval of existing types in the library are an important aspect of the design and implementation of new classes. The purpose of this work is to design and develop a tool that suport part of the design and implementation of classes process helping the recovery of software components from libraries.

AUTOMAÇÃO DO PROCESSO DE PROJETO E IMPLEMENTAÇÃO
DE CLASSES EM AMBIENTES ORIENTADOS POR OBJETOS

por

Mark Alan Junho Song
Roberto da Silva Bigonha

1 Orientação por Objetos

As metodologias atuais de análise, projeto e programação estão se baseando no paradigma da orientação por objetos utilizando os conceitos de objetos, classes, polimorfismo, herança e outros, com o objetivo de promover a produção de componentes de software com alto grau de reúso.

Segundo Meyer [BM88], um projeto orientado por objetos corresponde à construção de sistemas como uma coleção estruturada de **tipos abstratos de dados**. Como classes são mecanismos que possibilitam implementar tipos abstratos de dados (*TAD*), subentende-se que cada módulo implemente uma abstração, ou seja, um conjunto de estruturas de dados com as operações realizadas sobre as mesmas e suas propriedades.

Note que se exige uma *coleção*, ou seja, que as classes sejam definidas de forma a serem reutilizadas em aplicações para as quais não foram inicialmente projetadas. Além disso, é necessário que a coleção seja *estruturada*, isto é, que exista uma relação entre os vários *TAD* evitando-se desta forma a criação de classes isoladas não candidatas a reúso.

2 Etapas de Construção

2.1 Determinando os Objetos

Um aspecto importante a qualquer metodologia orientada por objetos é a determinação dos objetos e classes que compõem o modelo e a fatoração dos mesmos para a elaboração das diferentes hierarquias.

A identificação dos objetos pode ser feita, segundo Yourdon [EY94], analisando a especificação e extraindo os substantivos que são bons indicadores da presença dos mesmos na aplicação. Se um objeto é identificado isto significa que possui atributos e operações significativas.

2.2 Definindo Classes

Identificados os objetos, a próxima etapa é a definição das classes que descrevem aspectos semelhantes dos objetos identificados.

Note que nesta etapa o projetista pode se deparar com diversas dificuldades. Se certos atributos e operações são repetidos em um certo número de classes, eles certamente descrevem uma abstração que não tinha sido previamente identificada. Pode valer a pena criar uma nova classe que contenha tais características para que outras classes possam usá-las.

Algumas características podem ser armazenadas como atributos ou, por decisão de projeto, calculadas quando necessária. Define-se, neste caso, uma nova operação para a classe a partir de uma característica inicialmente identificada como atributo.

Um dos erros mais comuns é a definição de classes que são simplesmente processos encapsulados. Ao invés de representar tipos de objetos, estas classes representam funções encontradas por decomposição procedimental. É evidente que estas classes devem ser removidas do projeto.

Outro erro freqüente é a definição de classes desnecessárias [BM88]. Tome como exemplo a seguinte sentença: *A posição de um avião*. Identificam-se neste caso as classes *posição* e *avião* ou *posição* é meramente uma característica de *avião*?

Se posição pode ser visto como um *TAD*, com um conjunto de operações e propriedades significativas, tais como: medidas de erros, conversão entre sistemas de coordenadas, distância a uma outra posição, etc então *posição* é um sério candidato a classe. Caso contrário bastaria definir em *avião* 3 atributos reais que representem sua posição no espaço.

2.3 Identificando as Dependências

Uma extensão do passo anterior é a determinação das relações existentes entre as várias classes. Em geral, as relações são resultantes das dependências estabelecidas por suas interfaces. Se uma classe *Circulo* disponibiliza uma operação *Centro* que retorna o centro do circulo como um objeto do tipo *Ponto*, então um cliente de *Círculo* precisa conhecer a interface de *Ponto* para usar adequadamente a classe *Círculo*. Obviamente nem todas as dependências são visíveis na interface. Se uma classe *Lista* declara um membro privado do tipo *Arranjo* então clientes de *Lista* não precisam, e nem deveriam, conhecer a interface de *Arranjo* para usar a classe *Lista*.

Note que o projetista precisa conhecer todas as relações sejam elas visíveis ou não nas interfaces. Alterações em uma classe A que permite o acesso de B à sua representação,

pode implicar em alterações também em *B*.

2.4 Definindo as Hierarquias

Após a definição das classes e de suas relações, a próxima etapa é a fatoração de seus aspectos comuns organizando-as em hierarquias.

Novamente o projetista se depara com novos problemas. Características comuns devem ser descritas em uma superclasse e removidas de suas subclasses. Além disso, é preciso garantir que as relações estejam corretas. Dado as classes *avião* e *asa*, *asa* é um herdeiro de *avião* ou um de seus componentes? Parece óbvio que *avião* tem *asa* como um de seus componentes, uma *asa* não pode ser vista como um *avião*. Mas ao se considerar as classes *Lista* e *Pilha*, uma *Pilha* é uma *Lista* com operações especiais de inserção e retirada ou uma *Pilha* contém um membro do tipo *Lista*?

Cabe ao projetista alterar as relações identificadas incorretamente. Se a idéia é , por exemplo, apenas aproveitar a funcionalidade da classe então composição é a melhor solução.

2.5 Conclusão

A definição das classes de um sistema é um processo iterativo envolvendo uma série de etapas. Cada passo pode alterar as considerações efetuadas em passos anteriores obrigando que antigas etapas sejam refeitas com as novas informações disponíveis. Se cada etapa do processo for realizada apenas uma única vez, sem se preocupar com os passos anteriores é improvável a elaboração de hierarquias e a criação de classes candidatas a reúso.

Necessitam-se, desta forma, ferramentas que auxiliem o projetista no processo de projeto e implementação de classes permitindo que as alterações no projeto sejam efetuadas sem gerar grandes impactos.

3 Estado da Arte

O'Brien [PDM87] ressalta que ao se trabalhar com o paradigma da orientação por objetos é necessário adquirir conhecimento em pelo menos 4 áreas distintas: 1) a metodologia a ser usada, permitindo a formalização das especificações de uma dada aplicação; 2) o ambiente de programação e suas ferramentas de suporte; 3) a linguagem utilizada e seus recursos, e 4) a biblioteca de classes a ser usada.

Booch [GB94] salienta que um ambiente orientado por objetos deve oferecer um conjunto de ferramentas que permitam: 1) um sistema gráfico que suporte uma notação de

projeto onde alterações na representação implique em alterações na implementação e vice-versa; 2) a navegação pela hierarquia de classes visualizando métodos, atributos, suas relações de dependências etc; 3) uma gerência de biblioteca recuperando de forma eficiente componentes utilizando diferentes critérios de avaliação.

Ao se analisar os ambientes de programação como *Visual C++*, *Eiffel*, *Delphi* e outros percebe-se a existência de ferramentas que permitem a navegação pela hierarquia de classes, a visualização das relações de dependências, a definição de classes via edição etc.

Como, em geral, as linguagens só permitem que novas classes sejam criadas a partir de especializações de classes existentes, se o projetista identifica aspectos comuns em uma série de classes a única solução para estabelecer a nova relação é a edição das classes alterando seu código fonte. Este é um processo por demais restritivo pois implica em alterações em todas as subclasses da que foi generalizada.

Meyer [BM88] assume que a especialização é um processo normal de projeto onde se mantém intactas as superclasses, ao passo que generalizações são vistas como correções de omissões no projeto. Necessita-se que uma ferramenta suporte especializações assim como generalizações evitando que sejam feitas correções tardias de grande impacto no projeto.

Além disso, as ferramentas existentes nos ambientes de programação exigem que o projetista ou conheça os componentes para reúso, ou se disponha a efetuar uma pesquisa tediosa pela biblioteca de classes. Lembre-se que reutilizar um componente só tem sentido se o custo de localizá-lo é menor que o de criá-lo novamente.

A fim de evitar tais problemas e privilegiar o reúso, propõem-se então uma nova ferramenta que permita:

- suportar uma notação simplificada de projeto: classes, métodos, atributos, asserções, propriedades e hierarquias;
- a recuperação de componentes pela assinatura ou propriedades previamente definidas;
- a navegação pela hierarquia visualizando as classes, sua posição na hierarquia, suas relações de dependência, membros herdados e definidos, etc;
- a inclusão de classes por especializações e generalizações de classes existentes;
- a geração de protótipos de módulos que sirvam de modelo para edição;
- a manutenção automática dos protótipos gerados permitndo que alterações na interface da classes sejam refletidas no seu correspondente arquivo fonte;
- a edição da classe possibilitando efetivar sua implementação;
- a incorporação de alterações via edição através da compilação do seu arquivo fonte;
- a manipulação de membros e classes possibilitando a organização em hierarquias, a definição de seus membros sejam de dados ou funções, a movimentação e cópia de

classes e membros entre diferentes hierarquias etc.

4 Ferramenta Proposta

Figure 1: Class Designer.

Class Designer, exibida na Figura 1, é uma ferramenta de suporte que automatiza parte do processo de projeto e implementação de classes em Ita [MT95].

Ela possibilita estruturar classes de uma aplicação a partir das especificações fornecidas pelo projetista. Classes podem ser especializadas ou generalizadas. Membros de dados e funções podem ser associados às classes. Pode-se remover atributos e métodos ou mesmo classes definidas incorretamente sem que seja necessário a edição de seu código diretamente.

A ferramenta é um aplicativo Windows de fácil uso. Possui uma barra de *menus* com opções padrões para gerência de documentos (*File*), organização de janelas filhas na janela principal (*Windows*) e auxílio on-line (*Help*). O item *View* permite diferentes visualizações da hierarquia de classes. Uma barra de *status* é apresentada na parte inferior da janela principal exibindo informações para o usuário.

Basicamente trabalha-se com 2 tipos de documentos: 1) os de projeto, contendo as informações das classes e arquivos gerados e 2) os de arquivo fonte. Isto significa que apenas 2 tipos de janelas filhas são gerenciadas pela ferramenta: janelas de projetos e janelas com arquivos fontes.

Uma janela de projeto exibe 6 visões diferentes de um mesmo documento:

1. A hieraquia de classes. As hierarquias são exibidas como árvores onde um nó ancestral é visto como superclasse e nós descendentes como suas subclasses. Um nó especial denominado *Root* conecta as diferentes hierarquias dando origem a uma floresta de classes, é a raiz das diferentes hierarquias de classes.
   Uma classe selecionada nesta janela é exibida em destaque. Tal classe é denominada **ativa**. As demais janelas exibem informações referentes à classe ativa.
2. Propriedades. Exibe as propriedades definidas para a classe.
3. Membros herdados. Exibe os membros de dados ou funções que foram herdados ao longo da hierarquia.
4. Membros definidos. Exibe os membros que foram definidos ou redefinidos na classe.
5. Lista de uso. Exibe as classes que referenciam a classe ativa. Note, pela Figura 1, que a classe *linkable* referencia *linked_list* pois a declara como friend. Este fato não é facilmante observado analisando apenas a hierarquia de classes.
6. Lista de dependência. Exibe uma lista com as classes das quais a classe ativa depende.

4.1 Operações

O primeiro passo na elaboração das hierarquias é a determinação de um conjunto "flat" de classes que representem os objetos da aplicação. Uma vez definido o conjunto, organizam-se as classes em hierarquias estabelecendo as relaçoes existentes entre as mesmas.

Inserção

Class Designer possibilita que classes sejam inseridas em um projeto e posteriormente organizadas em hierarquias. A opção de *inserção* possibilita incluir classes sem que seja necessário definir seus métodos ou atributos. Com isto, consegue-se que o projetista se concentre apenas na determinação inicial dos objetos/classes que devem compor o sistema.

Este comando pode ser visto também como uma operação de especialização. As classes são inseridas sempre como subclasses da que está ativa. Se uma classe é "subclasse" de *Root* ela é então a base de uma nova hierarquia.

Class Designer verifica a entrada de dados da classe gerando um protótipo de módulo consistente com a linguagem.

Remoção

Durante o processo de organização das hierarquias podem ser detectadas classes definidas desnecessariamente. Lembre-se que o projetista irá definir, inicialmente, todas as classes que julgar importante.

A operação de *remoção* de classes permite corrigir erros de projeto através da eliminação de classes e arquivos associados. Este comando possibilita a eliminação, de forma definitiva, de uma classe específica ou de toda a hierarquia que a tenha como raiz. Cabe ao projetista escolher a opção que julgar mais adequada: 1) eliminar a classe e toda a sub-árvore associada ou 2) reestruturar a antiga hierarquia associando todas as subclasses diretas com a superclasse da que foi removida.

Movimentação/Cópia

Frequentemente deparam-se com situações em que uma classe é definida incorretamente em uma hierarquia devendo ser removida e inserida em outra.

Tome como exemplo uma hierarquia em que a classe base é um *Polígono*. Definem-se inicialmente *Triângulo* e *Ponto* como classes derivadas. Reta é definida como descendente de *Ponto*.

Após a definição dos métodos, nota-se que *Ponto* e *Reta* não devem ser definidas como herdeiras de *Polígono*. Não se aplica, por exemplo, o método *Calcula_Área* para pontos e retas. *Ponto* e *Reta* embora sejam partes de um *Polígono* não são polígonos. Necessita-se, desta forma, de uma operação que possibilite a remoção da classe e posteriormente sua inserção em uma nova hierarquia. Note que, neste caso, basta "movimentar" a classe *Ponto* para a nova hierarquia. Parece razoável supor que ao se deslocar uma classe deseja-se movimentar toda a hierarquia que a tenha como raiz.

A implementação de uma classe pode ser copiada para um módulo do sistema, utilizando a opção de *cópia* definida na barra de ferramentas. Este comando pode ser usado quando se verifica que uma classe só existe para facilitar os serviços definidos por outra, situação muito comum quando se está definindo componentes de software [SB91].

4.2 Generalizações

A inclusão de classes como descrita anteriormente é vista como especialização. Este processo permite que novas (sub)classes sejam definidas a partir de (super)classes acrescentando ou especializando serviços. Uma ferramenta que só suporte especializações exigirá que a hierarquia de tipos seja construída de forma *top-down*: a especialização requer que a superclasse exista antes da elaboração de suas subclasses.

No processo de definição das hierarquias reconhece-se normalmente os aspectos particulares antes de se perceber as similaridades entre as várias classes. Necessita-se desta forma de um mecanismo que possibilite a definição de superclasses a partir de subclasses. Generalização é este mecanismo.

Class Designer incorpora uma opção de generalização. Esta operação irá permitir que o projetista crie novas classes para incorporar elementos comuns identificados posteriormente, definindo, desta forma, hierarquias mais elaboradas.

4.3 Controles de Manipulação de Membros

Inserção

Após a definição das classes e possivelmente uma organização inicial das hierarquias associam-se métodos e atributos às mesmas.

Class Designer permite a inclusão de membros através do controle de *inserção* da barra de ferramentas. Membros são inseridos como públicos passando a fazer parte da interface da classe.

A escolha desta abordagem possibilita que o projetista se concentre apenas nos aspectos essenciais da classe sem, por enquanto, se procupar com os aspectos de implementação.

Note que após a inclusão dos membros deve-se reavaliar a organização inicialmente proposta. Se, como exemplificado anteriormente, definiram-se as classes *Triângulo* e *Ponto* como herdeiros de *Polígono*, ao se associar o método *Calcula_Área* a polígonos constata-se um erro na elaboração desta hierarquia.

Todas as assinaturas são verificadas automaticamente pela ferramenta. Assinaturas incorretas serão rejeitas.

Remoção

Métodos e atributos podem ser removidos da classe porque foram definidos incorretamente ou por se decidir mudar a representação.

Considere a classe *List* que define um método *Get_Count* informando a quantidade de

elementos da lista. Pode-se por questão de eficiência, por exemplo, substituir a função *Get_Count* por um atributo público *number* que mantém esta informação. Lembre-se que em Ita um atributo público é somente para leitura.

Movimentação

Ao organizar a hierarquia de classes depara-se com situações onde se percebe que um método da classe estaria melhor representado se estivesse definido na superclasse ou em algum outro ancestral. Lembre-se que uma classe deve definir um conjunto de operações que sejam genéricos o suficiente para serem reaproveitados por outras classes.

Considere uma hierarquia constituída por *Polígono* e sua subclasse *Triângulo*. Suponha que se defina inicialmente o método *Move* em *Triângulo* que o desloque de sua posição original.

Ao se definir a classe *Retângulo* derivada de *Polígono* nota-se que um um novo método em *Retângulo* ou então "mover" o que está definido em *Triângulo* para *Polígono*. A última opção certamente gera hierarquias de classes mais elaboradas.

Movimentam-se métodos e atributos com a opção de *movimentação* da barra de ferramentas. No caso de membros funções move-se também sua implementação externa.

Cópia

Membros podem ser copiados utilizando a opção de *cópia*. Este controle pode ser usado quando se identificam classes que tenham aspectos semelhantes mas não são vistas como subclasses de uma classe em comum.

Tome como exemplo uma classe *Lista* e suas operações de *Inserção*, *Remoção*, *Pesquisa* e etc. Ao se definir uma classe *Fila* pode-se organizá-la em uma hierarquia onde, por exemplo, *Fila* é vista como herdeiro de *Lista*. Outra solução seria definir uma nova hierarquia a partir de *Fila*.

Note que diversas operações em *Fila* são "semelhantes" à de *Lista*. Com o controle de *cópia* o projetista consegue definir um esqueleto para a classe *Fila* a partir de *Lista*. Basta editá-la posteriormente modificando as operações que julgar necessário.

4.4 Controles de Compilação

Class Designer mantém um conjunto de opções de compilação para validar as implementações das classes sem que seja necessário recorrer ao ambiente de programação. Além disso, permite incorporar novas classes ao projeto através da compilação de seu correspondente arquivo fonte.

Dentre as opções definidas pode-se compilar apenas uma classe (*Compile*), todas as classes do projeto (*Build*) ou apenas as que estão desatualizadas (*Make*).

Uma classe A está desatualizada se: 1) A versão compilada de A for anterior a edição dos seu correspondente fonte. 2) As classes das quais A depende estão desatualizadas. 3) As classes das quais A depende foram recompiladas após a compilação de A.

Procura-se, ao definir estas operações, que a ferramenta seja o mais independente possível do ambiente em que será instalada. O projetista não precisa recorrer ao ambiente de programação para compilar as classes de um projeto.

4.5 Edição de Classes

Class Designer permite a edição e gerência de múltiplos arquivos fontes. Cada classe define um arquivo de implementação (*.ita*) e um arquivo cabeçalho (*.ih*). Somente arquivos de implementação podem ser editados já que os arquivos de cabeçalho são gerados automaticamente pelo compilador.

A inclusão de um editor reforça a independência da ferramenta em relação ao ambiente. Além disso, possibilita que o projetista "salte" rapidamente da janela que contém o projeto para uma janela contendo as implementações de uma classe.

Para se editar um arquivo fonte seleciona-se a classe e posteriormente a opção de edição ou então dá-se um clique duplo no nome da mesma. Uma janela filha é criada contendo as definições da classe.

O editor trabalha com arquivos ASCII. Isto permite que os arquivos de implementação possam ser abertos externamente, se desejado, com diferentes editores.

Para que as alterações efetuadas em um arquivo possam ser refletidas no projeto deve-se compilar o mesmo. Este processo garante que as modificações sejam interpretadas corretamente. Class Designer efetuará todas as alterações necessárias na interface da classe e nas hierarquias.

Uma operação comum durante a elaboração das classes é a manutenção de suas interfaces. Se a mudança a ser efetuada for somente na interface pode-se utilizar uma opção especial da barra de ferramentas. Permite-se ao se selecionar tal controle corrigir a interface da classe e atualizar implicitamente seu arquivo de implementação sem que seja necessário editar o mesmo.

A última opção da barra de ferramentas permite obter informações a respeito dos arquivos que foram gerados para a classe. Estas informações são necessárias quando se manipulam os arquivos externamente.

5 Recuperação de Componentes

O objetivo da orientação por objetos é possibilitar que componentes de software possam ser reusados mesmo em aplicações para as quais nã o tenham sido inicialmente projetados. Isto impõe problemas de representação e recuperação dos candidatos: Que informações representar? Como adquirir tais informações? Como o projetista consultará o repositório de dados a procura de componentes?

Pode-se usar basicamente duas abordagens distintas [RY91]: 1) Recuperação de componentes (RC). Obtém-se as informações a partir do fonte, analisando a distribuição das palavras no texto. Nenhuma informação semântica é usada nem o documento interpretado. 2) Domínio Específico (DE). Obtém-se as informações por meio de sistemas especialistas que são sensíveis ao contexto e geram respostas adaptadas à experiência do projetista.

Uma biblioteca de classes permite a integração de ambas as abordagens. A documentação da classe permite o uso de técnicas RC, ao passo que a própria estrutura hierárquica, a aplicação de conhecimento adicional para pesquisa e recuperação eficiente.

No trabalho desenvolvido, as informações das classes especificadas pelo projetista servirão como informações pré-codificadas para a recuperação do componente. A compilação de arquivo fonte produz informações para a classe definida externamente. Usa-se ainda as propriedades como um recurso adicional neste processo.

Recuperam-se, através destas opções, classes que apresentem propriedades idênticas ou assinaturas equivalentes às da classe ativa. Este processo pesquisa por **toda** a hierarquia de tipos recuperando componentes candidatos a reuso.

5.1 Propriedades

Propriedades [PP95] são pares **(id, valor)** que permitem especificar as características da implementação. *Valor* é uma constante associada a *id*. São introduzidas pelo projetista e não aparecem no código *ita* implementado para a classe. Considere a seguinte classe:

```
class fixed_list < T > {
  private:
    int nb_elem;
    int position;
    array<T> buffer;
  public:
    init (int n) {
      buffer   = new array<T> (n);
```

```
      position = 0;
      nb_elem  = 0;
    }
  ..............
}
```

Pode-se definir ((estrutura, lista), (tamanho, fixo)) como as propriedades que caracterizam a implementação da classe *fixed_list*.

Convém ressaltar que propriedades definidas na superclasse **não** são herdadas por suas subclasses. Toda classe tem que definir suas propriedades se estas vão ser usadas posteriormente na busca de componentes reusáveis.

5.2 Recuperando Componentes por Propriedades

Pode-se recuperar um componente pelas propriedades associadas à classe. Considere (p_1, p_2 ... p_n) as propriedades da classe A e (q_1, q_2 ... q_m) as de B. Recupera-se uma classe B a partir das propriedades definidas em A se para cada p_i ($1 \leq$ i $\leq$ n) existir um q_j ($1 \leq$ j $\leq$ m) tal que $p_i = q_j$.

Dado, por exemplo, as classes:

- *list* com propriedade (estrutura, lista) e
- *fixed_list* com propriedades ((estrutura, lista), (tamanho, fixo))

então *fixed_list* será recuperada quando uma pesquisa por propriedades for efetuada a partir de *list*.

6 Protótipos de Membros

Um protótipo de membro é uma tupla que permite identificar um membro da classe. Se o membro é de dados a tupla é formada pelo **tipo** e **nome** do atributo, ou seja é o par (*Tipo, Nome*).

```
Dado a classe:

class  A < T > : C < T >
{
   .........
```

```
   public:
      int  x;
      void f (int w);
   .........
}
```

o protótipo do membro de dado *int x* é dado por (int , x).

No caso de membros funções, a tupla é composta pelo **tipo de retorno**, **nome da função** e sua **lista de parâmetros**. A lista de parâmetros é um conjunto de protótipos, onde parâmetros são identificados como os membros da classe.

Neste caso, o protótipo da função *f* é a tupla (void, f, { (int, w) }). A notação **{...}** define listas de elementos indicados.

7 Protótipos de Classes

Um protótipo de classe é uma tupla que permite identificar uma classe definida no projeto. É composta por:

- um nome, único na hierarquia;
- uma lista de parâmetros genéricos e
- uma lista dos protótipos dos membros da classe.

O protótipo da classe *A* é a tupla:

```
( A, {T}, { (int, x), (void, f, { (int, w) }) } )
```

8 Assinaturas de Membros

Seja *SigM* uma função que receba um protótipo de membro de dado retornando uma tupla com o tipo do membro de dado ou o tipo de retorno e uma lista com os tipos dos parâmetros do membro função ordenada por tipos.

Considere o exemplo anterior. Sabendo que o membro de dado *int x* tem como protótipo P_1 = (int, x) então *SigM*(P_1) = *(int)*. Analogamente, se P_2 é o protótipo para a função *f* então *SigM*(P_2) = *(void, { (int) })*.

Denomina-se a tupla retornada por *SigM* como a **assinatura do membro**.

9 Assinaturas de Classes

Seja *SigC* uma função que receba um protótipo de classe retornando uma tupla com: a lista de parâmetros genéricos e uma lista com a assinatura dos membros ordenada por tipo e valor de retorno.

`Dado a classe:`

```
class  A < T > : C < T >
{
   ............
   public:
      int x;
      void f (float w, int r);
   ............
}
```

denominando P_A o protótipo da classe A tem-se:
P_A = *(A , {T} , { (int, x), (void, f, { (float, w), (int, r) }) }).*
Logo:
SigC(P_A) = *({T}, { (int) , (void, { (float) , (int) }) }) .*
Denomina-se a tupla retornada por *SigC* como a **assinatura da classe**.

10 Equivalência de Classes

Considere S_A a assinatura da classe A e S_B a assinatura da classe B. Diz-se que a classe A é equivalente ($\approx$) a classe B se $S_A = S_B$.

`Considere as seguintes classes:`

```
class A < T >
{
  ...........
  public:
    int x;
    void f (float w, int r);
}
```

```
class B < T >
{
  ...........
  public:
    int K;
    void t (int m, float n);
}
```

onde:

M_1 = int x,
M_2 = int k,
M_3 = void f (float w, int r),
M_4 = void t (int m, float n).

Tem-se os seguintes protótipos de membros:

P_1 = (int, x),
P_2 = (int, k),
P_3 = (void, f, { (float, w), (int, r) }),
P_4 = (void, t, { (int, m), (float, n) }).

Como:

SigM(P_1) = (int),
SigM(P_2) = (int),
SigM(P_3) = (void, { (int), (float) }),
SigM(P_4) = (void, { (int), (float) }),

então, a assinatura da classe A é dada por:

S_A = *({T}, { (int) , (void, { (float) , (int) }) })*

e a assinatura da classe B dada por:

S_B = *({T}, { (int) , (void, { (float) , (int) }) }).*

Como:

$S_A = S_B$ então $A \approx B$.

10.1 Recuperando Classes pela Assinatura

Seja *Sel(A)* uma função que retorne um subconjunto dos membros da classe A a partir de uma seleção especificada pelo projetista e, *Members(B)* a função que retorne o conjunto dos membros da classe B.

Considere *Gen(A)* a função que retorne a lista dos parâmetros genéricos da classe ordenada por tipo (se o parâmetro for irrestrito é indicado por *IR*).

Dado a classe:

```
class A < T, E:B, Y >
{
  ..........
  public:
    int x;
    void f (float w, int r);
}
```

então *Gen(A) = (B, IR, IR)*.

Considere S_A a assinatura de uma classe A, e S_B a de B **candidata a reúso**. Uma classe B é recuperada da biblioteca se:

1. $S_B \approx S_A$ **ou**
2. $S_B \not\approx S_A$ **(A e B não genéricas), mas** *Members(B)* $\supseteq$ *Sel(A)* **ou**
3. $S_B \not\approx S_A$ **(A e B genéricas), mas** *Gen(B) = Gen(A)* **e** *Members(B)* $\supseteq$ *Sel(A)*.

Exemplo:

Seja A a classe ativa e B uma classe da biblioteca, S_A e S_B suas respectivas assinaturas onde:

```
class A < T, D:C >
{
  .......
  public:
    int a;
    float f (int x);
}
```

```
class B < W:C, K >
{
    .......
    public:
      int c;
      float k (int x);
      float g (int w);

}
```

Supondo que:

- M_1 = int a;
- M_2 = float f (int x);
- M_3 = int c;

- M_4 = float k (int x);
- M_5 = float g (int w);

e que *Sel(A)* = { *int a, float f (int x)* } então por (3) temos:

- $S_B \not\approx S_A$,
- *Gen(B)* = *Gen(A)* e
- *Members(B)* $\supseteq$ *Sel(A).*

logo B é uma classe que será recuperada quando uma **pesquisa por assinatura** for efetuada a partir de A.

11 Considerações Finais

Class Designer é uma ferramenta cuja atividade principal é a definição de classes a partir das especificações fornecidas pelo projetista ou obtidas pela compilação de um arquivo que implemente a classe.

Codificou-se a ferramenta em C++ usando o ambiente de programção *Visual C++*. É uma aplicação MDI suportando a manipulação de diferentes tipos de documentos.

Procurou-se dentro do possível seguir o padrão *MS-Windows* [MS92] facilitando, desta forma, o uso da ferramenta por usuários que já dominem aplicativos para o mesmo.

References

[BM88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall Inc., 1988.

[EY94] Edward Yourdon. *Object-Oriented Systems Design.* Prentice Hall Inc., 1994.

[GB94] Grady Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[MS92] Microsoft Co. *The Windows Interface - An Aplication Design Guide.* Microsoft Co., 1992.

[MT95] Marco Túlio de Oliveira Valente. *Projeto e Implementação da Linguagem de Programação Orientada por Objetos Ita.*

[PDM87] Patrick D. O'Brien, Daniel Halbert, Michael Kilian. *The Trellis Programming Environment.* OOPSLA'87, pp. 91-102, december 1987.

[PP95] Patrick Parot. *Mécanisation de la Réutilisation de Composants Logiciels: approches et outils.*

[RY91] Richard Helm, Yoëlle S. Maarek. *Integrating Information Retrieval and Domain Specifc Approaches for Browsing and Retrieval in Object-Oriented Class Libraries* OOPSLA'91, volume 26, number 11, pp 47-61, november 1991.