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

Mark Alan Junho Song[1]
Roberto da Silva Bigonha[2]
Mariza A. S. Bigonha[3]

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
30161-970 Belo Horizonte MG
Brasil

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 present a tool designed and developed to support part of the design and implementation of classes helping the recovery of software components from libraries.

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. Este trabalho apresenta o projeto e implementação de uma ferramenta, Class Designer, desenvolvida para automatizar parte do processo de projeto e implementação de classes auxiliando na recuperação de componentes das bibliotecas.

Keywords: Class Designer, Object-oriented programming, Class process.

[1]email: mark@dcc.ufmg.br
[2]email: bigonha@dcc.ufmg.br
[3]mariza@dcc.ufmg.br

1 Introdução

As metodologias atuais de análise, projeto e programação estão baseando-se 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 [BM97], 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 e 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*.

Um aspecto importante para qualquer metodologia orientada por objetos é a determinação dos objetos e classes que compõem o modelo e a fatoração da funcionalidade 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.

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. Em vez 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 [BM97]. 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* três atributos reais que representem sua posição no espaço.

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 *Círculo* disponibiliza uma operação *Centro* que retorna o centro do círculo 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.

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 incorretas identificadas. Se a idéia é, por exemplo, apenas aproveitar a funcionalidade da classe então composição é a melhor solução.

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.

2 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 quatro 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*, *Java* [GC97] 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 [BM97] 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 for menor que o de criá-lo novamente.

A fim de evitar tais problemas e privilegiar o reúso projetou-se e implementou-se uma nova ferramenta, **Class Designer**, que permite:

- 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 permitindo que alterações na interface da classes sejam refletidas no seu correspondente arquivo fonte;
- a edição da classe possibilitando efetivar sua implementação;
- 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.

Class Designer [MA96] foi originalmente projetada e implementada para operar com a linguagem Ita [MT96], [MB96]. Entretanto, dada a popularidade da linguagem Java [GC97] e considerando que a mesma é uma linguagem robusta, portátil e mais simples que as outras linguagens orientadas por objetos decidiu-se, como uma segunda etapa do trabalho, portar **Class Designer** para Java. Nesta nova versão **Class Designer** contém as mesmas facilidades implementadas na primeira versão acrescida de uma facilidade muito importante que consiste na incorporação de alterações via edição por meio da compilação do seu arquivo fonte em Java.

3 Visão Geral do Funcionamento de Class Designer

Class Designer é um aplicativo Windows de fácil uso. Ele possui um menu principal, uma barra de ferramentas e facilidades para acessar biblioteca de classes. Um menu é a linha que aparece logo abaixo da barra de título, como pode ser visto na Figura 1, cujo objetivo é controlar as operações que são executadas na janela principal.

3.1 O Menu Principal

O menu de Class Designer possui quatro grupos de palavras: `File, View, Window` e `Help`. Esses nomes quando abertos apresentam uma série de opções para seleção. Por meio deles a ferramenta controla as atividades da janela do aplicativo, como por exemplo, a abertura de novas janelas filhas. Pode-se ainda gerenciar documentos e escolher um modelo de visualização de classes dentre os permitidos pela ferramenta.

O menu `File` agrupa os comandos de manipulação de projetos e arquivos fontes. Um projeto é um documento gerado pela ferramenta que contém informações a respeito das classes especificadas pelo projetista. Cada classe definida em um projeto tem associado um arquivo

Figura 1: Class Designer.

fonte com extensão .java contendo sua implementação. O menu `File` permite ao projetista comandar ações que atuem na janela ativa, seja ela a principal ou uma janela filha. A comunicação entre o projetista e a ferramenta se dá por meio de caixas de diálogos.

O menu `View` destina-se a apresentar os comandos que atuam sobre as janelas filhas modificando a forma como as classes, definidas no projeto, são apresentadas. Por exemplo, geram diferentes visões de uma mesma hierarquia. Uma barra de *status* é apresentada na parte inferior da janela principal exibindo informações para o usuário. Ao se iluminar um comando do menu, uma descrição resumida da operação é exibida na barra de *status*.

Como, basicamente Class Designer trabalha com dois 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 seis 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*. Este fato não é facilmente observado analisando apenas a hierarquia de classes.
6. Lista de dependência: exibe uma lista com as classes das quais a classe ativa depende.

O menu `Window` é utilizado para acionar os comandos de redimensionamento e posicionamento automático das janelas filhas permitindo organizá-las na janela principal. Class Designer segue o padrão usado pela maioria das aplicações windows para exibição e organização de janelas.

O menu `Help` é usado para auxiliar, *on-line*, o projetista. Atualmente apenas informa a versão corrente da ferramenta.

3.2 A Barra de Ferramentas

Class Designer utiliza uma barra de ferramentas para ter acesso às operações disponíveis em um aplicativo ou um acesso rápido a algumas opções do menu principal. A barra de ferramentas é formada pelos seguintes controles: controle de documentos, controle de classes, generalizações, controles de manipulação de membros, controles de compilação e edição de classes. O controle de documentos possibilita um rápido acesso às operações relativas à gerência de documentos. As próximas seções apresenta cada um desses controles.

3.2.1 Controles de Classes

O primeiro passo na elaboração das hierarquias é a determinação de um conjunto de classes que representem os objetos da aplicação. Uma vez definido o conjunto, organizam-se as classes em hierarquias estabelecendo as relações 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 e 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 por meio 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].

3.2.2 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.

3.2.3 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 rejeitadas.

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.

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 possuir um conjunto de operações que sejam genéricas o suficiente para serem reaproveitadas 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.

3.2.4 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*).

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. A Seção 4.11 descreve a lógica de compilação.

3.2.5 Edição de Classes

Class Designer permite a edição e gerência de múltiplos arquivos fontes. Cada classe define um arquivo de implementação (*.java*) 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 via o compilador embutido em `Class Designer`. 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.

3.3 Acesso à Biblioteca de Classes

O processo de definição das hierarquias envolve o conhecimento de relacionamentos entre os diversos componentes: classes bases, derivadas, relações de inclusão, etc. Class Designer usa as informações especificadas no projeto asssim como as geradas pelo compilador auxiliando o projetista na recuperação e visualização dos componentes.

3.3.1 Visões da Hierarquia

A Figura 1 mostrada na Seção 3 ilustra uma janela filha que contém um projeto. Ela é dividida em seis visões diferentes para um mesmo documento: hierarquia de classes, propriedades, membros herdados, membros definidos, lista de uso e lista de dependência. A descrição dessas visões foi apresentada na Seção 3.1.

3.3.2 Recuperação de Componentes

A recuperação de componentes pode ser feita utilizando como critério as assinaturas das classes e/ou as propriedades previamente definidas. Recuperam-se, por meio dessas opções, classes que apresentem propriedades ou assinaturas idênticas às da classe ativa. Escolhe-se a opção de recuperação de componentes na barra de ferramentas. Após a escolha das opções e a confirmação da operação gera-se uma lista com os componentes recuperados exibindo as propriedades e/ou assinaturas comuns. Pode-se optar em recuperar: comente as classes com propriedades idênticas às da classe ativa; classes com mesmas assinaturas; classes com mesmas propriedades e assinaturas.

4 O Projeto de Class Designer

Nesta seção descrevem-se a especificação de Class Designer e sua integração com o ambiente Java.

4.1 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*). Por exemplo, a classe em Java dada

```
class  A extends C {
   .........
   public int  x;
   public 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.

4.2 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 (se houver) e
- uma lista dos protótipos dos membros da classe.

Dado o mesmo exemplo exibido acima, o protótipo da classe *A* é a tupla:

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

4.3 Assinaturas de Membros

Seja *SigM* uma função que recebe um protótipo de membro de dado e retorna 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 lexograficamente pelos nomes dos tipos.

Considere o exemplo apresentado na Seção 4.1. 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.**

4.4 Assinaturas de Classes

Seja *SigC* uma função que receba um protótipo de classe retornando uma tupla com: a lista de parâmetros genéricos (no caso de Ita) e uma lista com a assinatura dos membros ordenada por tipo e valor de retorno. No caso de Java a lista de parâmetros genéricos pode ser omitida.

```
Dado a classe:
class  A extends C  {
   ............
```

```
   public int x;
   public void f (float w, int r);
   ............
}
```

denominando P_A o protótipo da classe A tem-se:

P_A = *(A , { (int, x), (void, f, { (float, w), (int, r) }) } ⊕ Members(B))*.

onde:

A ⊕ B = A ∪ {y ∈ B | y ∉ A } e *Members(B)* uma função que retorna o conjunto dos membros da classe `B`.

Logo:

SigC(P_A) = *({ (int) , (void, { (float) , (int) }) })* .

Denomina-se a tupla retornada por *SigC* como a **assinatura da classe.**

4.5 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 $S_A = S_B$, i.e., se S_A tem exatamente os mesmos elementos que S_B.

`Considere as seguintes classes:`

```
class A {                          class B  {
  ............                           ............
  public int x;                          public int K;
  public void f (float w, int r);        public 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 = *({ (int) , (void, { (float) , (int) }) })*

e a assinatura da classe B dada por:

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

Como:

$S_A = S_B$ `então` $A \approx B$.

4.6 Propriedades

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

```
import java.util *;
class fixed_list  {
   private int nb_elem;
   private int position;
   private vector buffer;
   public init (int n) {
      buffer   = new vector( );
      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.

4.7 Recuperação de Componentes

O objetivo da orientação por objetos é possibilitar que componentes de software possam ser reusados mesmo em aplicações para as quais não tenham sido inicialmente projetados. Isto impõe problemas de representação e recuperação dos componentes em bibliotecas: 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.

Neste trabalho, as informações das classes especificadas pelo projetista servem como informações pré-codificadas para a recuperação do componente. A compilação de arquivo fonte produz informações para a classe definida externamente. Usa-se ainda o conceito de propriedades como um recurso adicional neste processo.

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

4.7.1 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)*.

```
Dado a classe:
class A  {
    ...........
    public int x;
    public void f (float w, int r);
}
```

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:

- $S_B \approx S_A$ **ou**
- $S_B \not\approx S_A$ **mas** *Sel(A)* $\subseteq$ *Members(B)*.

`Exemplo:`

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

```
class A  {                class B   {
  .......                    ........
  public int a;              public int c;
  public float f (int x);    public float k (int x);
}                            public 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 (2) temos:
- $S_B \not\approx S_A$,
- *Members(B)* $\supseteq$ *Sel(A)*

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

4.7.2 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*, mas *list* não seria recuperada se a pesquisa fizesse parte de *fixed_list*. Convém ressaltar que uma propriedade é vista internamente como uma seqüência de *strings*. *Strings* diferentes são tratados pela ferramenta como proriedades diferentes.

4.8 Generalizações

Generalização é o mecanismo que possibilita a construção de superclasses a partir de um conjunto de subclasses que apresentem características comuns. Class Designer só permite a generalização de subclasses que tenham um mesmo ancestral direto. O projetista que deseja criar uma nova classe por generalização precisa primeiro garantir estas exigências.

4.9 Geração de protótipos de Módulos

Class Designer associa a cada classe definida no projeto um módulo de implementação. Identifica-se o mesmo pelo nome da classe e por sua extensão .java. Um módulo contém as declarações da classe servindo de base para edição. Qualquer alteração na classe é automaticamente refletida no módulo.

A linguagem Java permite a declaração de mais de uma classe por arquivo fonte, ou módulo de compilação. Class Designer, contudo, adota a política de definir apenas uma classe pública por módulo. Como uma classe pode ser copiada entre módulos ela é sempre inserida, neste processo, como um componente privado do mesmo.

A implementação de membros funções ocorre fora da definição da classe. Gera-se para cada método não postergado uma implementação com o corpo vazio e com pré/pós condições desde que especificados. A definição da classe gerada contém apenas protótipos de funções e declarações de membros de dados. A ferramenta controla todas as definições necessárias por meio de marcas especiais que delimitam uma dada informação. Ressalta-se que estas marcas não devem ser removidas ou alteradas. Qualquer tentativa de atualização da classe poderá falhar se a mesma ou alguns de seus componentes não for identificado corretamente no módulo. Pode-se editar uma classe incorporando novos membros. Sua compilação pode resultar em alterações na interface.

4.10 Um Analisador para Declarações Java

Uma classe é composta, do ponto de vista da ferramenta, de uma seqüência de declarações Java. Toda especificação fornecida como entrada é validada por um analisador sintático construído a partir da gramática de Java. Possibilita-se dessa forma que Class Designer seja o mais independente possível do compilador gerando protótipos consistentes com a linguagem. Esta abordagem possibilita também que erros sejam diagnosticados e corrigidos antes da compilação do correspondente fonte.

4.11 Compilação

Class Designer disponibiliza três opções para compilação: *Make, Build, Compile*.

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.

Usa-se uma lista de dependências para a determinação das classes desatualizadas. A data do arquivo indica sua última modificação.

4.12 Implementação Dependende do Compilador

No projeto da ferramenta foi considerado importante a possibilidade de incorporar classes, na biblioteca de trabalho, a partir do seu arquivo fonte. Assim, Class Designer permite a compilação de arquivos definidos externamente.

Uma classe ao ser compilada gera informações que são armazenadas em uma lista de descritores de classes. Esta lista permite que a ferramenta incorpore a nova classe no projeto. Um descritor de classes é composto: pelo nome da classe; por seu nível na hierarquia; uma indicação se é abstrata; a lista de parâmetros genéricos (se Ita); a lista de classe friends (no caso de Ita); a lista de seus membros de dados e ou funções.

Se uma classe é definida internamente, com o auxílio de Class Designer, gera-se um protótipo de módulo contendo suas definições. Note que este módulo pode ser editado externamente. Novamente, a compilação será usada para consistir os dados das classes já definidas com seus correspendentes arquivos. A compilação pode, dessa forma, resultar na alteração da interface da classe.

A estrutura de dados e as rotinas necessárias para que possam ser efetuadas as devidas alterações no compilador são inseridas como ações semânticas no arquivo de entrada para o gerador de análise sintática.

5 Conclusão

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.

Class Designer foi codificada em C++ usando o ambiente de programação *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.

Notadamente, Class Designer é uma ferramenta que facilita o desenvolvimento de sistemas. Uma das principais vantagens em relação a outras ferramentas como o *Good* de Eiffel [BM88], *Class Wizard* de C++ [MS93b] ou o *Browser* de Delphi é a possibilidade de se definir a hierarquia de classes para posteriormente efetivar sua implementação. Nos ambientes pesquisados não encontramos uma ferramenta que realmente permitisse o projeto de classes. Em geral, dispõe-se de ferramentas que permitem visualizar as classes da biblioteca, as relações de dependência, os atributos e métodos. *Good*, por exemplo, permite inserir ou remover classes em uma hierarquia, mas não permite definir classes por generalização. Todos os membros são definidos por edição. Erros de projeto são corrigidos editando-se as classes envolvidas. *Class Wizard* por sua vez só permite definir métodos e associar variávies para os controles definidos na classe. Ele opera com classes e mensagens para objetos derivados da biblioteca MFC. A procura por componentes é feita com o auxílio de um *browser* que gera uma base de dados para o projeto a partir da compilação de classes. Se a base de dados não for gerada, não se tem acesso as informações das classes definidas pelo programador.

Referências

[BM88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall Inc., 2nd Edition, 1997.

[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.

[MT96] Marco Tulio de Oliveira Valente.*Projeto e Implementação da Linguagem de Programação Orientada por Objetos ITA*, Tese de Mestrado, DCC-ICEX, UFMG, 1996.

[MB96] Marco Tulio de Oliveira Valente e Roberto da Silva Bigonha, *A Linguagem de Programação ITA*, Relatório Técnico 005/96, Departamento de Ciência da Computação, ICEx, UFMG, Fevereiro de 1996.

[MA96] Mark Alan Junho Song .*Mecanização do Processo de Projeto e Implementação de Classes em Ambientes Orientados por Objetos*, Tese de Mestrado, DCC-ICEX, UFMG, 1996.

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

[MS93b] Microsoft Co. *Users Guides - Visual Workbench, App Studio.* Microsoft Co., 1993.

[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.

[GC97] Gary Cornell, Cay S. Horstmann. Core Java , Makron Books, 1997.