Construção e Geração de uma Hierarquia de Classes a partir de Código Java

Relatório Técnico do Laboratório de Linguagens de Programação - LLP010/99

Arthur Zambelli de Almeida
José Raphael de Souza Lamas
Mariza Andrade S. Bigonha
Roberto da Silva Bigonha

Resumo

Neste documento vamos apresentar os resultados da compilação de Class Design para Java. Class Design foi originalmente desenvolvido para a linguagem Ita por Mark Alan J. Song. A linguagem Ita foi projetada pelo Prof. Roberto da Silva Bigonha e implementada por Marco Túlio O. Valente e faz parte dos projetos desenvolvidos pelo Laboratório de Linguagens de Programação da UFMG.

Ita é uma linguagem orientada por objetos e possui características da linguagem C++. Class Desing é uma ferramenta criada com o intuito de automatizar parte do processo de projeto e implementação de classes, auxiliando o programador durante a consulta à biblioteca.

O trabalho que fizemos consistiu de duas partes: na primeira parte reaproveitamos toda a metodologia desenvolvida por Song e mudamos o alvo do objeto de Ita para Java. A segunda parte consistiu da construção da hierarquia de classes e validação de dados de entrada. Adaptamos também a parte gráfica do sistema. a escolha de Java como novo ambiente se deve ao fato de que está sendo cada vez mais usada por um grande número de pessoas e a interface neste ambiente é bastante amigável.

Sumário

1 Introdução

Como a orientação por objetos [4] está fundamentada em objetos e classes, um aspecto essencial em qualquer metodologia orientada por objetos [4] é a determinação de classes e objetos que irão compor o modelo. De acordo com [2] uma das dificuldades encontradas durante o projeto de um sistema orientado por objetos é a obtenção dos objetos e, ou classes e também a fatoração das propriedades comuns para a construção das várias hierarquias. Com isso inúmeras questões surgem, por exemplo, como determinar os objetos; como é feita a comunicação entre os vários objetos; que serviços oferecer; a que classe pertencem; como determinar os serviços de novas classes para reaproveitar os serviços das classes já existentes etc.

O projeto de pesquisa proposto baseia-se no desenvolvimento de uma nova versão para a ferramenta *Class Designer* [2] desenvolvida no DCC-UFMG. `Class Designer` é uma ferramenta de suporte para a linguagem ITA [1]. Ela foi criada com o intuito de automatizar parte do processo de projeto e implementação de classes, auxiliando o programador durante a consulta a biblioteca. O trabalho cujo resultado relatamos neste documento compreendeu duas partes: na primeira parte, a idéia foi reaproveitar toda a metodologia usada por Song [2] e mudar o alvo do objeto, ou seja, gerar Java [3] ao invés de ITA [1]. A segunda parte, consistiu na construção da hierarquia de classes e validação dos dados de entrada. A parte gráfica do programa consistindo na apresentação gráfica da hierarquia também foi adaptada. A escolha de Java [3] como novo ambiente se deve ao fato de que está sendo cada vez mais usado por um grande número de pessoas e a interface neste ambiente é bastante amigavél.

Este relatório está organizado da seguinte forma. A Seção 2 apresenta as motivações e objetivos deste projeto. A Seção 3 apresenta a revisão da literatura. Inicialmente são mostradas as principais características da ferramenta `Class Designer`[2] seguida da apresentação da linguagem Ita[1], e finalmente os pontos mais relevantes da linguagem Java[3], a linguagem usada como objeto em nosso trabalho.
As Seções 4 e 5 mostram como o projeto proposto foi desenvolvido. Especificamente, a Seção 4 apresenta o resultado da engenharia reversa de um código fonte em Java, que representa uma funcionalidade não existente no programa original. A Seção 5 apresenta como foi feita a construção da hierarquia de classes, a validação dos dados de entrada e a adaptação da apresentação gráfica das hierarquias.

A Seção 6 apresenta as conclusões deste trabalho, seguida da bibliografia.

2 Motivações

Uma das principais vantagens do paradigma da programação orientada por objetos [4] é o desenvolvimento de programas a partir da reutilização de código. Consequentemente, o conhecimento dos tipos existentes na biblioteca constitui um aspecto importante no projeto e implementação de novas classes, uma vez que a organização de um programa se baseia nos tipos que ele manipula. Classes permitem a implementação de tipos abstratos de dados em linguagens orientadas por objetos [4] .
Song [2] em seu trabalho ressalta que ao se trabalhar com o paradigma de orientação por objetos [4] é importante ter conhecimento não só da metodologia a ser usada mas também: do ambiente de programação e suas ferramentas de apoio; da linguagem utilizada e das bibli-

otecas de classes a serem usadas. Com estas idéias e com o objetivo de solucionar as questões expostas na Seção 1 ele projetou e implementou uma ferramenta de suporte que automatizou parte do processo de projeto e implementação de classes para a linguagem ITA [1]. Esta ferramenta oferece recursos para: definir assinatura de classes; incluir classes em hierarquias; incluir, remover ou alterar dados em uma determinada classe via interface; editar classe; acessar a biblioteca para consultar, exibir as hierarquias de classe, suas propriedades e membros; visualizar a relações de dependências entre classes, etc.

3 Revisão da Literatura

3.1 Introdução

"O projeto orientado por objetos é caracterizado por um conjunto de técnicas distintas que visam modelar o sistema de software como uma coleção estruturada de tipos abstratos de dados."[4]

As metodologias de análise, projeto e programação baseadas no paradigma da orientação por objetos[4] utilizando os conceitos de objetos, classes, polimorfismo, herança, generalização e outros, tem como objetivo promover a produção de softwares com alto grau de reuso.
Um projeto orientado por objetos corresponde a construção de sistemas como uma coleção estruturada de tipos abstratos de dados.
A orientação por objetos[4] permite que esses novos tipos abstratos de dados(TAD's) sejam adicionados a uma linguagem e posteriormente reutilizados em outras aplicações para as quais eles não foram inicialmente projetados.
A classe é o mecanismo que permite implementar estes TAD's. Com ela é possível criar novos tipos a partir dos já existentes definindo novas classes. Com isso 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.
O conceito de generalização corresponde a uma espécie de relacionamento "é-um" entre duas classes, na qual uma classe pai, também chamada de superclasse, pode ser sub-dividida em diversas outras, chamadas subclasses.
Ao se trabalhar com o paradigma da orientação por objetos[4] é necessário ter conhecimento em pelo menos 4 áreas distintas:

- a metodologia a ser usada.
- o ambiente de programação e suas ferramentas de suporte.
- a linguagem utilizada e seus recursos.
- a bibliioteca de classes a ser usada.

Uma consulta a biblioteca de classes já existente pode levar a reutilização de classes previamente construídas, o que facilita muito o trabalho do desenvolvedor além de contribuir para uma das principais características da orientação por objetos[4] que é o reuso de código.

3.2 Ferramenta Class Designer

Um sistema para projetar classes é uma ferramenta capaz de montar toda a hierarquia de classes, listando todos os seus atributos, métodos e heranças, a partir do código fonte.

Como na orientação por objetos existe a possibilidade de se utilizar objetos em outras aplicações, para as quais eles não foram originalmente definidos, é necessário ter controle sobre os atributos e métodos de todos os objetos criados, pois agora, eles podem ser úteis de novo. A ferramenta *Class Designer* foi projetada com este propósito. Ela pode recuperar uma lista de todas as classes já criadas, e junto com elas, todos os seus atributos métodos, heranças etc, ou seja, todas as informações referentes àquela classe.

Além de ser capaz de desmontar toda a hierarquia de classes a partir do código fonte, o *Class Designer* consegue, também, gerar o código fonte a partir da hierarquia de classes, o que torna muito fácil o processo de efetuar manutenções em um programa.

Na próxima seção serão abordadas as características de utilização do programa, bem como uma pequena apresentação das funcionalidades principais contidas dentro do programa.

3.2.1 Utilização do Programa Class Designer

O programa *Class Designer* é um programa para *windows*, e, como tal, faz uso de janelas e menus para fazer a interface com o usuário. Ele é um aplicativo MDI (multiple documents interface), ou seja, ele permite que mais de uma janela filha esteja aberta dentro de uma janela principal de aplicação ao mesmo tempo. Na barra de menus da aplicação existem quatro menus. São eles:

- File: Ítem de menu dedicado à gerência de documentos.
- View: Ítem de menu que permite diferentes visualizações da hierarquia de classes.
- Window: Ítem de menu que facilita a gerência das janelas filhas que por ventura existam.
- Help: Ítem de menu que chama o auxílio on-line do sistema.

Como a ferramenta foi desenvolvida para trabalhar com projetos de programas e códigos fonte de programas, existem, logicamente, somente dois tipos de documentos de entrada válidos, são eles: (1) os de projeto, contendo informações sobre as classes e arquivos gerados para este projeto e (2) os de arquivo fonte.

Independente de qual o tipo de documento que está aberto em um dado momento da execução do programa, as mesmas funcionalidades e visões são disponibilizadas. As visões oferecidas pelo *Class Designer* consistem em maneiras de agrupar a informação relativa a um determinado objeto de modo a permitir ao usuário visualizar mais facilmente uma determinada característica do objeto. Ao todo, o *Class Designer* oferece seis possíveis visões, que podem ser acessadas simultaneamente ou individualmente através do menu "view" que já foi abordado anteriormente. As visões oferecidas pelo programa seguidas de uma pequena explicação de sua funcionalidade são:

- Hierarquia de classes: nesta visão a hierarquia de classes é exibida sobre a forma de árvore, onde cada nodo da árvore é uma classe, e caso um nodo esteja ligado a um nodo pai, isto significa que o nodo ancestral é uma superclasse e os nodos descendentes são suas subclasses. Nesta visão existe um nodo ancestral de todos, o nodo "root", este

nodo não tem nenhuma correlação com o projeto, ele existe somente para reforçar a interligação entre as classes de um determinado projeto. Nesta visão, ainda, é possível selecionar um determinado nodo da árvore, este nodo passa a ser o nodo ativo, e em todas as outras visões serão exibidas somente informações relativas a este nodo ativo. O nodo ativo é também conhecido como classe ativa.

- Propriedades: exibe as propriedades definidas para a classes ativa.
- Membros herdados: exibe as propriedades ou métodos que foram herdados ao longo da hierarquia de classes para a classe ativa.
- Membros definidos: exibe as propriedades e os métodos que foram definidos ou redefinidos na classe ativa.
- Lista de uso: lista as classes que referenciam a classe ativa.
- Lista de dependências: lista as classes das quais a classe ativa depende.

Como o programa não é somente uma ferramenta para a visualização das informações relativas a hierarquia de classes de um dado projeto, mas é também uma ferramenta que permite a manutenção, ou até a criação de classes, ele permite que o usuário interaja em qualquer uma das visões e faça as alterações desejadas.

Para permitir uma completa autonomia do código fonte durante a interação com o *Class Designer*, todas as operações que podem ser desejáveis dentro de um projeto feito sobre o paradigma da orientação por objetos são implementadas dentro do programa. Ele permite que se modifique ou que se crie operações, generalizações, métodos, atributos de uma determinada classe.

Além de ser aplicável em projetos, o *Class Designer* ainda pode ser empregado para a reutilização de componentes que é uma outra importante facilidade oferecida pela orientação por objetos. Na área de objetos reutilizáveis ele permite que sejam recuperadas somente algumas classes de um projeto (a partir do código fonte do projeto) e que estas classes sejam inseridas dentro de outro projeto. Para isso, ele traz já implementadas diferentes políticas de recuperação de objetos, desde a recuperação de objetos por propriedades até a recuperação de objetos por assinatura.

Depois que um projeto já foi devidamente criado, ou modificado, o Class Design, pode gerar o seu código fonte, e, ainda, compilá-lo, caso seja do interesse do usuário. A única restrição quanto a compilação do código fonte é a de que existe a necessidade de haver um compilador de ITA já instalado na máquina, pois o *Class Designer* não possui um compilador de ITA implementado dentro dele. O que ele faz quando o usuário pede uma compilação é executar uma linha de comando no *shell* que passa ao compilador ITA todas as diretivas e arquivos fonte necessários para compilar o projeto e gerar o arquivo executável.

3.3 Linguagem ITA

Ita é uma linguagem orientada por objetos projetada pelo Prof. Roberto da Silva Bigonha [6] e implementada por Marco Tulio Valente[1]. Ela representa um super conjunto de C, como C++, porém teve como objetivo introduzir em C conceitos de OO[4] como herança, polimorfismo e encapsulamento de dados. Ela se baseia nas linguagens como Eiffel[7] e Oberon-2[8] além de prover outras facilidades. Estes conceitos de OO[4] permitem produzir software com

alto grau de reuso e correto por meio de um estilo de programação por contrato. Além disso a semântica de Ita[1, 6] favorece o uso e o entendimento por não possuir um excesso de recursos. Esta seção irá tratar do que foi acrescentado a linguagem C tomando como base que o leitor já conheça a mesma.

3.3.1 Conjunto de Inteiros

Ita[1, 6] possui um tipo de constante a mais, que é a constante para conjuntos de inteiros não negativos. Ela também possui um tipo de dado `set` usado para declarar esse mesmo conjunto. As operações que se aplicam a esse tipo são:

- união(+)
- interseção(*)
- diferença(-)
- pertinência(in)
- diferença simétrica(|).

Exemplos:

- Constante de conjunto de inteiros: {0,1,2,3}
- Tipo de dados: set x={0,2,4,5} + y;

3.3.2 Classes

Classes são usadas em Ita[1, 6] para implementar tipo abstrato de dados(TAD's). Ao defini-las desta maneira as classes passam as ser dotadas de um mecanismo de encapsulamento visando a programação em grande escala. A classe define então um tipo que será usado para declarar os objetos que são as instâncias da classe. A implementação dos métodos de uma classe pode ocorrer no próprio corpo da classe ou fora dele. Quando ela é feita por fora há apenas um protótipo para o método no corpo da classe. Ela é necessária no caso da classe ser mutuamente recursiva. O operador de qualificação(::) indica a qual classe tal método pertence. Em Ita existem dois especificadores de acesso: *public* e *private*. Um deles vigora até a definição do outro. O inicial é o *private*. Os métodos ou atributos dentro do escopo *private* são visíveis apenas no corpo da classe enquanto o do especificador *public* possui o mesmo escopo do nome da classe, porém para atributos públicos o acesso é apenas para leitura. Em Ita quando um objeto é declarado não quer dizer que ele já está instanciado. Os operadores *new* e *delete* tratam respectivamente de instanciar e remover o objeto da área de mémoria que ele ocupa.

Uma classe pode possuir dois tipos de métodos especiais chamados de inicializadores e finalizadores. Métodos inicializadores são chamados implicitamente pelo método `new` e suas principais funções são a de inicializar a área de mémoria para o objeto e garantir a validade de seu invariante. Uma lista de parâmetros pode fazer parte do método `new`, para ser utilizado pelo método inicializador. Já o método finalizador não apresenta parâmetros algum e é chamado pelo operador delete para que se libere a área de memória do objeto.

Em Ita existe o conceito de classes *friends* que atenuam o conceito de encapsulamento uma vez que uma classe permite que todas as classes *friends* a ela possam utilizar de seus métodos privados. Porém para que ela possa alterar seus atributos públicos e privados ela tem de ser uma subclasse *friend*. Existe também o conceito de classes genéricas que possuem tipos como parâmetros. Ela é importante pois permite que o mesmo método possa ser aplicado para diferentes tipos. Estes parâmetros podem ser restritos ou irrestritos. Os parâmetros irrestritos são aqueles que aceitam qualquer tipo de parâmetro. Somente operações de comparação, atribuição e duplicação serão feitas. Os tipos restritos no entanto aceitam somente subtipos de um tipo restringente. Com isso todas as operações deste tipo restringente serão aplicadas a referências genéricas.

Exemplos:

```
// Exemplos de classes, classes friends, inicializador e finalizador
class Conta Friend Pessoa{
int Saldo;
...
}
...

class Pessoa{ // Membros Privados
int Senha; // atributo
int RetornaSenha(....){.....} // método
.......
public: .... // Membros Públicos
int Identidade; // atributo
void Testa(...){.....} ; // protótipo de método

init(int,float){....} // inicializadora
finish(){.....} // finalizadora

AcessaSaldo(){
Conta MinhaConta= new Conta;
int meuSaldo=minhaConta.saldo; // Acessa o atributo privado de Conta já que ela é friend
.....
}
}
....

int Pessoa::Testa(...){.....} //implementação externa do método

....

Pessoa Maria= new Pessoa (30,65.5); //chama o método init
Maria.Testa(...);
delete Maria; // chama o método finish

....
```

```
// Exemplo de classe genérica
class Numeros <Tipo> {
....
Tipo Valor;
Tipo Calcula(Tipo Num) {....}
....
}
...

Numeros <int> Inteiros; // Classe que utiliza dos números inteiros para seus cálculos
Numeros <float> Reais; // Classe que utiliza dos números reais para seus cálculos
```

3.3.3 Passagem de Parâmetros

Toda a passagem de parâmetros em Ita[1, 6] é por valor. Para parâmetros do tipo classe passa-se por valor uma referência para o objeto.

Exemplo:

```
class Carro {
public:
int Chassi;
init(int Temp) {Chassi=Temp;} // inicializadora
void setChassi(int Temp) {Chassi=temp;} // atualiza o numero do chassi
}

void ZeraChassi (Carro Fiat) {Fiat.setChassi(0); }
.....
Carro Fiat= new Carro(345); //fiat.chassi=345
ZeraChassi(Fiat);
printf("%d
n",fiat.chassi) ; //fiat.chassi=0
....
```

3.3.4 Tipos de Funções

Funções em Ita[1, 6] podem ser polivalentes e polissêmicas. Funções polivalentes são aquelas que têm parâmetros formais de tipo classe ou genéricos. Para os parâmetros formais do tipo classe ela pode receber parâmetros formais dessa classe e de suas subclasses. Para os parâmetros do tipo genérico ela pode receber uma hierarquia de tipos dependendo do tipo de parâmetro genérico(restrito ou irrestrito). Polissemia é a capacidade de uma função possuir várias implementações sendo todas elas denotadas pelo mesmo nome. A decisão de qual implementação ser executada é avaliada na ordem:

- O número e tipo dos parâmetros das funções ;

- A forma do objeto denotado pela referência sobre a qual aplica-se a função;
- O estado do objeto denotado pela referência sobre a qual se aplica a função.

Uma função é distinguível da outra pelo seu número e tipo de parâmetros se ela for distinguível pela sua identificação. A identificação de uma função é o par

(nome-função, assinatura)

onde assinatura são os tipos de parâmetros. Duas identificações são iguais se o nome da função for o mesmo, o número de parâmetros for o mesmo e se para o mesmo parâmetro ou ele é igual ou um subtipo do outro. Funções polissêmicas por estado são aquelas de mesma assinatura mas com implementações variando dependendo do estado da classe. O especificador `state` tem a função de definir os estados de objetos de uma classe por meio de um conjunto de valores dos seus atributos. A avaliação é feita na ordem em que aparecem na definição da classe.

Exemplo:

```
// Exemplo de polissemia por estado e por número de parâmetros

class Clima{

int temperatura,umidade;
state frio=temperatura < 15;
state quente = temperatura > 30;
state umido = umidade > 65 ;

int CalculaGasto(int,float) at frio {.....}
int CalculaGasto(int,float) at quente {.....}
int CalculaGasto(int,float) at umido {.....}

....

// Polissemia por número de parâmetros
int CalculaPressaoAtm(int Altitude,int Latitude,int PressaoLocal) {....}
int CalculaPressaoAtm(int Altitude,int Latitude) {....}
} ....
```

3.3.5 Asserções

Ita[1, 6] tem o privilégio de possuir o uso de expressões booleanas, avaliadas em tempo de execução, que descrevem propriedades de trechos de código. A essas expressões dá-se o nome de asserções, que podem ser de três tipos:

- Invariante
- Pré-condição
- Pós-condição

O invariante é uma asserção associada a classe que deve ser válida no inicio e fim da execução de cada um de seus métodos. Ela tem que ser válida também para todas as instâncias da classe. A pré-condição é associada a um método da classe e ela tem que ser válida no inicio de sua execução. A pós-condição é associada a um método da classe e ela tem que ser válida no fim de sua execução. Métodos inicializadores e finalizadores não podem conter nem pré nem pós condições. Já o invariante é executado depois que o método inicializador foi executado e antes do finalizador o ser. Quando um método não contiver uma pré ou pós condição assume-se que ela será TRUE. Quando uma asserção é violada uma exceção será gerada. A ativação desta exceção transfere o controle para o bloco rescue associado a função na qual ocorreu a exceção. O bloco *rescue* corresponde a um trecho de código localizado no final de cada função que irá terminar de duas formas. Uma delas é executando até o final do bloco e nesse caso a função falha. O ponto de retorno dele será desviado para o bloco *rescue* da função chamadora e dai por diante até o bloco *rescue* da função *main*, caso eles também falhem. A outra opção é de se executar um *retry*. Nesse caso o corpo da função é executado novamente com a reavaliação da pré-condição e mantendo-se os valores correntes de suas variáveis automáticas. Ao se usar o *retry* a exceção é desativada.

Exemplo:
Class Salario{

int Valor;
minimo=130;
....
int CalculaBonus(int Saldo)
pre(Saldo > 130) //pré-condição

pos ((result > 0 || Valor=initial(valor) + 100)) //pós-condição

rescue{..... retry;.....} //bloco rescue de CalculaBonus

{.....}
.....
invariant: (valor > minimo) //invariante(Salário nao pode ser menor que o mínimo)
}
rescue{..... retry;.....} //bloco rescue da Classe Salario

3.3.6 Herança

Herança é o mecanismo utilizado em orientação por objetos[4] que permite construir classes por meio da especialização de outras classes já existentes. Uma classe Y é dita ser subclasse de uma outra classe X se Y for construída a partir de X, que é chamada de superclasse. Como uma classe, em Ita[1, 6], define um tipo, uma subclasse irá definir um subtipo de sua superclasse. Com isso um objeto de um subtipo pode ser usado onde seu supertipo for esperado. A subclasse herda de sua superclasse todos os métodos e funções com exceção dos inicializadores e finalizadores mas só tem direito a acessar os membros públicos. Ela pode também criar seus próprios membros assim como redefinir os que foram herdados. Apesar da subclasse

não herdar o método inicializador de sua superclasse ela pode definir que este método seja executado antes de seu próprio inicializador. O invariante da superclasse é automaticamente combinado ao invariante da subclasse através de um and lógico. Ao se redefinir um método ainda sim é possível utilizar o original da superclasse bastando para isso utilizar da palavra reservada `super` seguida de um ponto e do nome do método.

Exemplo:
Class Pessoa {
int Identidade;
int RetornaDados(int Codigo) {.....}

Public:
int Idade;
init (int Temp) { idade=temp;}
int Imposto(int) {.....}
....
}.....

Class Estudante:Pessoa {
int NumMatricula;
int RetornaCurso(int Dado) {.....}

Public:
int AnoEscolar;
init(int Grau,int idade):Pessoa(idade) {AnoEscolar=Grau;}
.....
}...

3.3.7 Polimorfismo

Ita[1, 6] possui referências polimórficas que podem denotar objetos de uma classe e de todas as suas subclasses. Com isso a referência pode ser do tipo estático ou dinâmico. Estático é a que ela apresenta no momento em que ela foi declarada. Tipo dinâmico é o tipo efetivo que ela apresenta em tempo de execução. Este tipo pode ser altrerado por uma atribuição ou uma passagem de parâmetro. Para testar qual tipo uma referência denota, foi criado em Ita[1, 6] duas construções. A primeira chamada de teste de tipo informa se o tipo dinâmico de uma referência é igual ou derivado de um certo tipo. A outra chamada de guarda de tipo é uma asserção que tem como objetivo garantir que uma certa referência possui um determinado tipo.

Exemplo:

A a; //tipo estático de a: A
B b=new B; //tipo estático de b: B
a=b; //tipo dinâmico de a: B
//a pode executar métodos da classe B

3.3.8 Classes Abstratas

Classes abstratas são aquelas criadas com o intuito de deixar o projeto mais correto, porém elas não possuem instâncias. Seus métodos só estarão implementados em suas subclasses. Estes métodos são chamados de postergados(deferred) e na classe abstrata apenas o seu protótipo está definido. No protótipo as pré e pós condições têm que estar definidas. Caso um método postergado não for implementado em sua subclasse uma violação ocorrerá.

Exemplo:
abstract Class X { .... // classe abstrata
deferred void f (int x) // método postergado
pre(x>0) pos(...); // inclui as pre e pos-condicoes
...
}

3.4 Linguagem Java

Hoje em dia, temos uma grande variedade de máquinas e sistemas operacionais diferentes sendo utilizados em todos os tipos de aplicações. Não existe uma padronização universal que defina o conjunto de instruções, seus parâmetros, etc para as máquinas e nem um padrão que defina métodos comuns de gerenciamento de memória, acesso a discos, estrutura de arquivos, etc. Para sistemas operacionais não é diferente. É impossível desejar que um mesmo programa execute em plataformas diferentes sem necessitar ser recompilado.

Fica muito difícil e caro para uma empresa comercial gerenciar versões de seus sistemas para diferentes plataformas. Como solução para estes problemas a linguagem Java foi criada. Seus principais objetivos são: (1) fornecer uma linguagem única multiplataforma, (2) fornecer uma melhor forma de interação com usuários via `internet`.

Como a linguagem Java é iminentemente voltada para `internet`, onde existe uma variedade quase infinita de plataformas querendo acessar a mesma informação, os criadores de Java se aproveitaram do `Browser` como instrumento de acesso que um internauta usa para acessar as páginas na internet. O `Browser` é um programa que permite a um usuário acessar páginas repletas de informações na `internet`. O que ele faz é interpretar o código `html` que existe na página que se deseja acessar, e, de acordo com as instruções contidas nesta página ele desenha a página. Com Java acontece básicamente a mesma coisa, ou seja, quando se deseja acessar uma página que contêm algum aplicativo em Java, o `browser` lê todo o arquivo em Java e depois começa a executar as instruções contidas neste arquivo.

As diferenças básicas entre Java e `html` são: enquanto html é uma linguagem estática, não prevê a passagem do tempo nem interações com o usuário, Java é uma linguagem mais próxima da realidade de programa. O fluxo de execução de um programa pode ser alterado de acordo com interações com o usuário.

O ambiente que permite que um programa em Java seja interpretado dentro de um `browser` é chamado de máquina virtual Java. Esta máquina virtual Java faz com que Java consiga ser independente de plataforma. A máquina virtual Java nada mais é do que um emulador de um processador que foi definido pelos criadores de Java. Este emulador possui um conjunto de registradores e um conjunto de instruções. A máquina virtual Java é uma facilidade oferecida pelos `browsers`, e ela tem que ser exatamente igual em todos os `browsers`.

Dessa forma, um programa que foi escrito em Java pode ser executado em qualquer plataforma desde que esta plataforma possua um `browser` e este `browser` possua a máquina virtual Java implementada.

3.4.1 Características da Linguagem Java

A linguagem Java é uma linguagem totalmente orientada por objetos, ou seja, em Java não pode existir um programa constituído por uma série de funções que são chamadas de dentro de uma função principal.

A semântica dos comandos em Java são muito semelhantes a semântica de C++, porém como Java é puramente orientada por objetos somente os comandos em C++ relacionados a criação, instanciação, chamada de propriedades e métodos de classes em C++ possuem correspondentes em Java. Além destas funcionalidades semelhantes as de C++ Java também possui eventos, o que facilita e muito a interação com o usuário.

Como Java é uma linguagem voltada para a internet, e é interpretada ao invés de compilada, o nome programa não é a denominação mais correta que deve ser aplicada a um aplicativo Java e sim "applet" (aplicativo). Isto ocorre porque um programa geralmente é a denominação dada ao código já compilado, pronto para ser executado, coisa que não ocorre dentro de um aplicativo em Java.

3.4.2 Tipos de Dados

Java[3] é uma linguagem fortemente tipada, ou seja, toda variável deve ter um tipo declarado. Há oito tipos primitivos em Java[3]. Os seis tipos numéricos são ilustrados na tabela abaixo.

Tipo	Armazenamento	Faixa
int	4 bytes	-2.147.483.648 a 2.147.483.647
short	2 bytes	-32.767 a 32.767
long	8 bytes	-9.223.372.036.854.775.808 a -9.223.372.036.854.775.807
byte	1 byte	-128 a 127
float	4 bytes	faixa aprox. -3,40282347E+38F(6-7 digitos significativos)
double	8 bytes	faixa aprox.-1,79769313486231570E+308(15 digitos significativos)

Os literais inteiros longos possuem um sufixo L, assim como os do tipo float possuem um sufixo F e os do tipo double, D. Os tipos de ponto flutuante seguem a especificação do IEEE 754. Além disso, os tipos inteiros não dependem da máquina em que eles estão sendo executados. As faixas para os vários tipos são fixas.
Além desses seis tipos númericos, Java[3] também apresenta um tipo booleano, *boolean*, e um tipo *char*. Ao contrário do tipo *string*, o tipo *char* usa aspas simples para indicar um caractere. Ele ocupa dois *bytes* permitindo a utilização de 65.536 caracteres. Ele usa um código denominado *Unicode* que engloba o código ASCII/ANSI, que só permite 255 caracteres.

3.4.3 Variáveis, Constantes e Atribuições

Java[3] requer que seja declarada uma variável. Isto é feito colocando o tipo em primeiro lugar seguido do nome. Tem que haver um ponto e virgula no final de cada declaração e não se pode usar uma palavra reservada de Java para um nome de variável. O nome deve começar

com uma letra e ser uma sequência de letras ou dígitos.
Uma variável pode ser inicializada explicitamente por meio de uma instrução de atribuição. Ela representa o nome da variável seguida por um sinal de igualdade e um valor apropriado para ela. É importante sempre inicializar a variável.
As conversões de variável são permitidas sem haver um conversor explicito para alguns casos de tipos numéricos. É permitido atribuir uma variável de um tipo que está a esquerda ao tipo da direita da seguinte lista:

byte -> short -> int -> long -> float -> double

As constantes só são permitidas para todos os métodos da classe, logo não há como criar uma constante local para um método.

3.4.4 Operadores

Os operadores aritméticos normais(+,-,* e /) são usados para adição, subtração, multiplicação e divisão, respectivamente. O operador / indica divisão inteira se ambos os argumentos forem inteiros ou ponto flutuante, caso contrário. Os operadores de incremento e decremento fazem parte de Java[3], de maneira semelhante a linguagem C. O operador x++, por exemplo, soma um ao valor atual da variável x. O operador x-, ao contrário, subtrai um do valor de x. Eles podem ser utilizados dentro de expressões, antes ou depois do nome da variável. Caso seja utilizado como prefixo do nome da variável, a operação é realizada antes. Já como sufixo da variável, a expressão considera o valor antigo da variável e só então depois realiza a operação. O uso desses operadores em expressões deve ser evitado.
O operador == é usado para comparar o valor de uma variável ou expressão. O operador != para a deigualdade. Os operadores de maior que(>) ou menor que(<), maior ou igual(>=), menor ou igual(<=) são também usados. Java[3] usa os operadores && e || para realizar um AND e OR lógico, respectivamente. Eles são avaliados em curto-circuito. Por exemplo, no caso de um AND, se o valor para a primeira expressão tiver sido determinado como falso, a segunda expressão não será calculada. Além dos operadores lógicos, Java[3] contém operadores que podem trabalhar diretamente com os bits, realizando um and(&), or(|), xor($\wedge$) e not($\sim$).

3.4.5 Strings

Strings são sequências de caracteres como "Hello". Java[3] não possui um tipo primitivo *string* interno. Ela possui uma classe predefinida, chamada *String*. É possível realizar diversas operações com as *strings*. Uma delas é a de concatenar uma ou mais *strings* utilizando o operador +. Com o método *substring*, é realizada a extração de uma *substring* dentro da *string* especificando para isso o seu tamanho e o seu inicio na *string* original. O método *length* fornece o número de caracteres de uma *string*.

3.4.6 Fluxo de Controle

Java[3] aceita instruções condicionais e *loops*, como qualquer outra linguagem de programação. As instruções condicionais podem ser da forma :

if (condição) instrução ;

if (condição) { bloco }

if (condição) instrução *else* instrução;

if (condição) { bloco1 } *else* { bloco2 } ;

Bloco corresponde a qualquer número de instruções que é cercado por um par de chaves. Os blocos definem o escopo de suas variáveis. No entanto, é possível declarar variáveis de nomes idênticos em dois blocos diferentes, que existam ao mesmo tempo.
Os *loops* podem ser determinados ou não. No caso dos *loops* determinados temos as seguintes opções: while (condição) { bloco } ;

do { bloco } while (condição) ;

No segundo caso ele executa o bloco e, só então, testa a condição.
Para os *loops* determinados temos :

for (instrução1; expressão1; expressão2) { bloco };

Para o caso de seleções múltiplas, utiliza-se a instrução *Switch*:

switch (escolha)

{ *case 1:*

....

break;

case 2:

.....

break;

...

default:

...

break;

}

Várias alternativas podem ser disparadas, pois a execução passa para outro *case*, a menos que uma palavra chave *break* o faça sair da construção switch inteira. A cláusula *default* é opcional.

Ao contrário de C e C++, Java[3] oferece uma instrução *break* rotulada que permite sair de múltiplos *loop* aninhados.

3.4.7 Classes e Objetos

Para a maioria das classes em Java[3], tem de ser criado um objeto relativo a ela. Este objeto tem de ter seu estado inicial especificado. A partir dai o objeto será utilizado. Para acessar os objetos, são definidas variáveis de objeto. Por exemplo:

Animal gato; //Variavel de objeto gato da classe animal

ela esta definindo uma variável de objeto gato, que pode ser referir a objetos do tipo Animal. É importante notar que esta variável não é um objeto, neste ponto ainda não se pode usar os métodos da classe animal. Para criar um objeto o operador *New* tem de ser usado:

gato = New Animal(); //Cria-se uma instância de animal

Com isso, os métodos da classe Animal podem ser usados. Se uma variável de objeto for atribuída a outra usando o sinal de igualdade, então ambas irão referir-se ao mesmo objeto. Basta definir uma variável objeto como *NULL* para que ela não se refira a qualquer objeto. Apesar de outras linguagens adotarem a passagem de parâmetros de objetos como de referência, já que objetos são referências, em Java[3] eles são passados por valor. Logo, métodos nunca podem mudar os valores de seus parâmetros.
A simples sintaxe para uma classe em Java[3] é:

class NomedaClasse
{ // definições dos recursos da classe
// métodos e campos da instancia
}
O par de chaves externo, define o código que irá compor a classe.
Tal como Ita[1, 6], Java[3] também possui modificadores de acesso, *public e private*. A palavra chave *public* indica que qualquer método que tenha acesso a uma instância desta classe, pode chamar o método. Já no caso da palavra reservada *private*, nenhum método localizado em outra classe poderá acessá-lo. Ela é extremamente útil quando se tem em uma classe métodos que foram desmembrados de métodos maiores e só são chamados por eles. Eles deveriam ficar escondidos dos usuários. Isto também se aplica para as propriedades, apesar de ser incorreto utilizar o especificador *public* com elas. Ao se realizar isto, qualquer parte do programa poderia ler ou modificar estes dados, anulando o princípio do encapsulamento.
Toda classe em Java[3], apresenta um ou mais métodos com o seu mesmo nome. Eles são utilizados para inicializar objetos de uma classe e podem ser diferidos apenas pelo número e tipo dos parâmetros. Estes métodos são chamados de construtores. Quando se inicializa um objeto, utilizando do operador *New* , o método construtor correspondente é invocado. O correspondente é aquele que combina exatamente os parâmetros fornecidos com os que ele possui. A capacidade de se realizar isto é chamada de sobrecarga.
O objeto pode ser destruído com o método *finalize()*. Ele é chamado antes do coletor de lixo varrer qualquer objeto.
Toda classe em Java[3], tem que ter um método main que será o principal e primeiro a ser

executado, depois de ela estar inicializada.

3.4.8 Pacotes

As classes em Java[3] podem ser agrupadas em uma coleção denominada pacote. Eles são convenientes para organizar o trabalho feito e separá-lo de outras bibliotecas de código. Os pacotes podem ser aninhados tal como subdiretórios em um disco rígido. O nome do pacote deve vir no topo do código da seguinte maneira:

package NomeDoPacote;

se este nome for omitido a classe é adicionada ao pacote *default* de Java[3].
Pode se utilizar as classes públicas de um pacote de duas maneiras. Uma delas é fornecendo o nome completo do pacote seguido do nome da classe, por exemplo java.util.MinhaClasse. A segunda maneira é muito mais prática. Basta utilizar a palavra reservada *import* seguida do nome do pacote. Ela é bastante flexível pois permite importar todas as classes de um pacote sem ter de ficar fornecendo para cada uma o seu nome completo. Por exemplo:

import java.util.*;

neste caso todas as classes deste pacote incluindo MinhaClasse serão importadas. Ele também pode ser utilizado para importar somente uma classe do pacote, porém não pode ser utilizado para se importar mais de um pacote. Os especificadores *public e private* são utilizados com classes também. Se uma classe for privada somente as classes dentro do mesmo pacote poderão acessá-la.

3.4.9 Herança

Java[3] permite o uso do conceito de herança entre classes, utilizando para isso da palavra reservada *extends*. Ao se definir uma subclasse basta fornecer depois de seu nome a palavra *extends* seguida do nome da superclasse a qual ela herdará os seus membros. Por exemplo:

class Filha extends Pai ....

O construtor das subclasses pode, antes de se inicializar, chamar o construtor da super classe, bastando para isso utilizar da palavra-chave *super*,
seguida dos parâmetros que deseja passar:

super(Parâmetro1, Parâmetro 2,...)

Para que uma classe não possa ter subclasses é utilizado a palavra-cave *final* ao se definir a classe, como se segue:

final class NomeDaClasse

3.4.10 Classes Abstratas

Classes abstratas, também estão presentes em Java[3]. Para que uma classe seja abstrata, é utilizado a palavra-chave *abstract* no momento de se definir a classe. Por exemplo:

public abstract class ClasseAbstrata {.....}

O objetivo de classes abstratas é o de tornar o projeto das classes mais claro. Muitas vezes o projetista só quer que, determinado método seja implementado em subclasses de uma superclasse. Com isso a palavra chave *abstract* pode ser usada para os métodos também, indicando que eles serão implementados mais adiante na hierarquia de classes.

3.4.11 Exemplo de Aplicativo em Java

Segue abaixo trechos de um aplicativo em Java que ilustra bem a orientação por objetos e demais funcionalidades da linguagem. A função `conta` conta o número de caracteres digitados pelo usuário.

```
import Java.io.*;

public class Contadora
{
public static void conta(Reader entrada) throws IOException
{
int cont = 0;
while (entrada.read() != -1)
cont++;
System.out.println(cont + "Caracteres.");
}
}
```

Este programa mostra a declaração de uma classe pública chamada "Contadora". Esta classe não possui nenhuma propriedade, somente um método chamado de "conta". O método conta recebe um parâmetro, a variável "entrada", que é do tipo "Reader". Dentro do método "conta" temos a declaração de uma variável inteira chamada de "cont" que é inicializada com zero. Depois entramos dentro de uma estrutura de repetição do tipo "while" que é análoga ao do C++. Dentro desta estrutura de repetição o que acontece é que enquanto existirem caracteres na variável "entrada" o contador será incrementado de um. Depois, quando não mais existirem caracteres na variável "entrada" o aplicativo imprime o número de caracteres que ele contou na tela.

Apesar de simples, este trecho de aplicativo ilustra muito bem o grau de orientação por objetos de Java, além de ilustrar também as semelhanças com C++.

3.4.12 Outro Programa em Java

```
import java.util.*;
import Teste.*;
```

```
public abstract Class Pessoa //Classe abstrata
{
public Pessoa(string Name) //Construtor
{ Servico ClasseTemp;
Nome=Name;
ID=Servico.GetID(Nome); //Classe Servico é definida no Pacote Teste
}

public static void main(String[] args)
{int i=0;
int anos=50;
Servico Temp;
int PesoIdeal;

for(i=0,i¡anos,i++)
{ if i=Idade then
{PesoIdeal=Temp.CalculaAumentoPeso(i);
break;
}
}
}

public abstract void CalculaSalario(double SalarioAtual); //Metodo postergado

// Propriedades
public string Nome;
private long ID;
public boolean Sexo;
private double Salario;
private int Idade;
public int Peso;

}

public final class Estudante extends Pessoa //Classe Estudante foi herdada de Pessoa
{
public Estudante(string Name) // Classe Estudante nao pode ter subclasses
{ super(Name);
}
}
```

4 Metodologia usada no Desenvolvimento do Módulo

4.1 Introdução

Como toda a implementação de software, antes de dar início à implementação propriamente dita é importante descrever as principais funcionalidades do programa e a partir destas funcionalidades definir a estrutura de dados a ser utilizada, uma vez que a mesma exerce um impacto decisivo sobre a qualidade e a funcionalidade do programa a ser implementado.

Este projeto não se trata de um novo sistema e, sim, de uma adição de funcionalidades em um sistema já existente. Assim sendo, existem dois pontos principais a serem lembrados antes de se começar a implementação do trabalho:

1. As novas unidades devem ser completamente encapsuladas, o que diminui o número de linhas de código a ser acrescentado no programa fonte original

2. A nova parte do programa deve interagir perfeitamente com o restante, utilizando a mesma estrutura interna de dados, e mantendo todas as funcionalidades anteriores.

Não devemos perder de mente que o objetivo final deste projeto é: (a) adicionar ao programa *Class Design* a capacidade de funcionar perfeitamente com Java, e, mais que isso, (b) que seja capaz de fazer a engenharia reversa de um programa já existente para dentro de uma hierarquia de classes. Esta seção descreve as decisões de projeto e resultado de (b). A parte (a) é descrita em 5.

4.2 Estruturas de Dados

Como a engenharia reversa nada mais é do que uma compilação do arquivo de entrada e a carga dos dados referentes à sua hierarquia de classes, a estrutura de dados criada para o desenvolvimento deste módulo é basicamente uma tabela de símbolos que controla escopo, permissão de acesso, herança, etc, para todos os seus símbolos. Esta tabela deve ser otimizada para a compilação. Além de ser otimizada para a compilação, esta tabela de símbolos também deve permitir a integração com o programa, pois ela contém todas as informações necessárias

para se carregar a estrutura interna de dados do programa. Segue abaixo a especificação da tabela de símbolos:

```
typedef struct properties *prop_apontador;
typedef struct params *param_apontador;
typedef struct metodos *met_apontador;
typedef struct classes *cla_apontador;

typedef struct properties {
char nome[60]; representa o nome
int tipo; representa o tipo
int modificadores; indica quais são os modificadores
cla_apontador tipo_aux; quando o tipo de dados for de um tipo diferente de um dos tipos
```

padrão Java
prop_apontador esq, dir; implementa lista duplamente encadeada
} propriedade;

typedef struct params {
char nome[60]; representa o nome
int tipo; representa o tipo
int modificadores; representa quais são os modificadores
cla_apontador tipo_aux; quando o tipo de dados for de um tipo diferente de um dos tipos padrão Java
param_apontador esq, dir; implementa lista duplamente encadeada
} parametro;

typedef struct metodos {
char nome[60]; representa o nome
int tipo; 0 - procedure, 1+ - funcao com o tipo
int modificadores; indica quais os modificadores
cla_apontador tipo_aux; quando o tipo de dados for de um tipo diferente de um dos tipos padrão Java
param_apontador parametros; apontador para a estrutura que armazena os parâmetros
prop_apontador variaveis; variaveis locais
met_apontador esq,dir; implementa lista duplamente encadeada
} metodo;

typedef struct classes {
char nome[60]; representa o nome
int modificadores; indica quais são os modificadores
prop_apontador propriedades; indica quais são as propriedades
met_apontador metodos; indica quais são os métodos
cla_apontador pai, filho, irmao; implementa lista duplamente encadeadae mais uma lista horizontal
} classe;

Nas estruturas descritas, existem alguns campos comuns a todas. Estes campos são aqueles responsáveis pelo armazenamento das principais características de cada elemento. O campo modificadores é o único campo cujo entendimento não é trivial. Os modificadores de um elemento são *static, private, protected, public* etc. Como mais de um modificador pode ser aplicado a um mesmo elemento, a melhor opção para este campo foi torná-lo um inteiro, sendo que cada modificador seria um inteiro potência de dois. Esta escolha foi feita para permitir que se descubra se um dado elemento possui ou não um dado modificador. Existem modificadores que são exclusivos, isto é, quando um deles for utilizado o outro não pode ser usado. Este estilo de armazenamento dos modificadores não garante isto, por isso, antes de inserir um novo modificador em um elemento deve existir uma rotina que verifica se este novo modificador é válido ou não dentro do elemento que ele está sendo inserido.

Além da tabela de símbolos, o programa também deve possuir uma estrutura para armazenar os erros. Esta estrutura deve contemplar todas as informações relativas aos erros que aconteceram na compilação de um programa, tais como, linha do erro, mensagem de erro, tipo de erro (erro fatal, warning), etc.

Outras estruturas de dados tiveram que ser criadas para contemplar todas as definições da linguagem Java, mas por não serem intrissecamente ligadas ao projeto e, sim, à implementação não irei discutí-las aqui, mas sim mais adiante, quando estivermos discutindo a implementação propriamente dita.

4.3 Implementação do Módulo para o Class Design

A implementação deste módulo do projeto *Class Design* em Java foi dividida em três partes, são elas:

1 - Analisador Léxico.
2 - Analisador Sintático e Semântico.
3 - Rotinas de transformação da estrutura interna de dados do módulo para a estrutura interna do programa *Class Design* e chamada das rotinas de atualização de tela, arquivos, etc.

Existe uma sequência lógica entre destas três fases, e ela foi obedecida dentro do processo de implementação. Isto é, primeiramente foi implementado o analisador léxico, em seguida o analisador sintático, e depois as rotinas de integração entre o *Class Design* e o módulo.

Na criação do analisador léxico foi utilizado um gerador automático chamado *flex32*. Na criação do analisador sintático também foi utilizado um gerador automático chamado *bison32*. Ambos os geradores empregados são da mesma família do *lex e yacc*, porém são versões para *DOS* ao invés de *Unix*, e a decisão de usá-los veio justamente pelo fato deles gerarem um código mais compatível com o compilador usado: *Visual C++ 1.0*. O analisador sintático e semântico foi desenvolvida junto, uma vez que, o programa gerador do analisador sintático oferece espaço para se definir as rotinas semânticas dentro do mesmo arquivo de entrada.

Nas seções seguintes abordaremos cada uma das três fases da implementação separadamente apresentando as metodologias de projeto empregadas e discutindo porque foram feitas.

4.3.1 Análisador Léxico

A fase de análise léxica tem como função ler o arquivo de origem e retornar para o analisador sintático os *tokens*.

Muito pouca codificação foi feita nesta fase. Como foi utilizado um gerador automático para gerar o analisador léxico, a codificação foi feita dentro do arquivo de entrada utilizado.

O arquivo de entrada para o gerador deve conter as produções léxicas, suas classificações e qual a ação a ser tomada quando se encontra um *token* daquele tipo. A ação padrão se resume em retornar para o analisador sintático qual o *token* encontrado.

As únicas diferenças em relação ao comportamento padrão surgem para o caso de constantes, tanto numéricas quanto literais, e no caso dos identificadores. As constantes tiveram um tratamento diferente porque antes de retornar para o analisador sintático qual o *token* encontrado, era necessário verificar se o valor condizia com o tipo do *token*. Por exemplo, se um *token* é do tipo float e tem mais casas decimais do que o permitido para aquele tipo de

dados. Caso seja encontrado um erro aqui, a estrutura de erro descrita em [?] é carregada com um erro de valor de variável.

Os identificadores também precisaram de um tratamento diferenciado. Isto aconteceu porque para todas as rotinas semânticas é necessário saber qual o identificador. Devido a isto, antes de retornar para o analisador sintático que o *token* encontrado é um identificador a palavra encontrada é inserida em uma tabela temporária. Esta tabela temporária existe porque um identificador pode ocorrer várias vezes dentro de um mesmo arquivo e a cada vez se trata de uma entrada diferente da tabela de símbolos. Este problema ocorre muito em linguagens orientadas por objetos, especificamente com definições de escopo e visibilidade tão marcantes como em Java. O fato desta tabela temporária poder armazenar mais de um valor ao mesmo tempo será discutida na Seção [?].

A tabela temporária foi implementada na realidade por uma lista encadeada. Sua estrutura de dados é a seguinte:

```
typedef struct temptabs {
char palavra[60]; representa qual é a palavra
temptabsapontador dir;
} TempTab;
```

A estrutura é bastante simples, ela armazena a palavra atual e qual o próximo elemento da lista. Além desta estrutura existem dois apontadores globais que armazenam o topo e o final da lista. Isto é feito para garantir que quando for utilizado um valor da tabela se utilize todos, e para facilitar a limpeza das palavras já lidas. Em conjunto com esta tabela existem duas funções, uma que insere um valor na tabela, e uma que retira um valor da tabela. A primeira é utilizada dentro do analisador léxico, já a segunda é utilizada dentro do analisador sintático, mas, como ambas são muito triviais elas não serão abordadas neste texto.

4.3.2 Analisador Sintático e Semântico

Este projeto se comporta exatamente como um compilador durante a fase da análise léxica, sintática e semântica, a única diferença surge pelo fato de que neste projeto não existe geração de código intermediário e nem código objeto. Isto ocorre porque o propósito deste projeto é a carga da estrutura de dados do programa *Class Design* com a hierarquia de classes definida dentro de um programa em Java e a verificação de um código fonte feito em Java. Para isso, só são necessários os analisadores léxicos, sintáticos e semânticos.

Como a criação do analisador sintático e semântico é feita em conjunto, esta é a fase mais importante para o projeto, pois é nesta fase que a hierarquia de classes do programa é montada, e também é nessa fase que o programa é verificado por erros sintáticos e semânticos, que constituem a maioria dos erros cometidos durante a implementação de um programa.

O programa que foi utilizado como gerador do analisador sintático e semântico recebe um arquivo de entrada com as produções gramaticais válida e as respectivas ações semânticas para cada produção. Como as rotinas semânticas costumam ser muito repetitivas, muitas das rotinas semânticas foram transformadas em chamadas de procedimentos, e, estes procedimentos, executam o processamento semântico.

Sintáticamente, não existiu a necessidade de criar nenhum procedimento além daqueles gerados pelo programa gerador. Agora, semanticamente, foi necessário que se criasse tudo,

uma vez que o programa gerador somente fornece uma forma de se entrar com as rotinas semânticas, não fornecendo nenhum suporte a elas.

Como o objetivo do módulo era somente possibilitar a engenharia reversa de um programa, e a sua verificação, todos os procedimentos semânticos criados são de natureza de validação e armazenamento de classes, métodos ou propriedades. Foram criados vários procedimentos semânticos com o intuito de serem bastante reutilizáveis. Existem procedimentos que inserem classes, métodos, propriedades e parâmetros, outros que validam alguns atributos, e, ainda, outros que pesquisam um elemento pelo nome e retornam apontadores para a estrutura caso o elemento procurado seja encontrado.

Além dos procedimentos semâticos, foi necessário a criação de procedimentos que tratam erros e tratam os *imports*. Os erros são tratados a partir de uma estrutura, que é composta da seguinte maneira:

```
typedef struct erros {
char desc[200]; representa a mensagem
char palavra[60]; representa a ultima palavra lida
int tipo; indica qual o tipo de erro
int linha, col; indica qual a linha e a coluna dentro do arquivo
erros esq,dir; implementa lista duplamente encadeada
} Erro;
```

Todos os atributos da estrutura de erros são claros, exceto o atributo tipo. Este atributo existe para implementar os dois tipos de erro: fatal, warning. Quando o erro é fatal a execução do módulo é interrompida e a mensagem é retornada ao usuário, já quando o erro é do tipo warning a execução não é interrompida, mas no final é exibida uma lista ao usuário com todos os erros encontrados e seus tipos. Além destes erros ainda existe o erro de sintaxe, este erro é tratado pelo próprio gerador e a mensagem retornada para o usuário é *"sintax error"*. Este erro é sempre fatal, e, dessa forma a execução do módulo é interrompida e é mostrado ao usuário a linha onde foi encontrado o erro sintático.

Os *imports* nada mais são do que uma forma para incluir um arquivo em Java. A forma usada para tratar os *imports* consiste de uma estrutura de dados que armazena o nome do arquivo a ser incluído e uma chamada recursiva da própria rotina de verificação sintática com outro arquivo como argumento. O único cuidado especial ao se efetuar esta chamada foi fazer com que todas as variáveis relacionadas a tabela de símbolos fossem globais, dessa forma, elas seriam válidas independente do empilhamento de rotinas sintáticas e de arquivos a serem lidos. Como este módulo não tem um caminho de procura de arquivo, caso o arquivo incluído não exista, nenhum erro nem warning é retornado ao usuário. Somente se for utilizado um objeto que tenha sido definido neste arquivo o usuário verá uma mensagem de erro, mas o tipo do erro será warning, pois dentro do contexto da criação de uma hierarquia de classes a partir de um código fonte em Java, não faz sentido nenhum incluir os objetos que já vem com a linguagem Java. Por exemplo, se dentro de uma aplicação existe uma propriedade de uma classe que é do tipo *string*, ao fazer a engenharia reversa neste arquivo, a classe *string* não deveria aparecer dentro da hierarquia de classes montada, pois ela não agrega nenhuma informação acerca do programa.

Existe, ainda, uma dificuldade para a criação deste módulo que não foi abordada neste texto. Ela está associada ao fato de que em Java é possível fazer uso de uma classe antes

de declará-la. Para resolver este problema, uma outra estrutura de dados foi criada. Esta estrutura armazena o nome do objeto e os parâmetros de seu construtor. Para cada inserção de uma nova classe é verificado se existe alguma referência a esta classe na estrutura de classes que ainda não foram declaradas mas já foram utilizadas, caso exista, basta verificar se a chamada do procedimento de construção de classe é válido e, caso seja, ótimo, senão um erro é retornado.

A inserção de elementos é bastante complexa. Ela deve levar em conta a herança de classes e os modificadores inclusive das classes ancestrais. Isto ocorre devido à existência em Java dos modificadores *abstract* e *final*. O modificador *abstract* significa que este método deve ser definido em todas as classes filhas da classe atual, e, que este método não está definido nesta classe. Já o modificador *final* é o oposto, ele significa que a definição atual de um método, propriedade ou classe, é a última na hierarquia, ou seja, esta classe não pode ter herdeiros, ou os métodos não podem sofrer *overload*. Todos estes aspectos são devidamente abordados dentro do módulo e corretamente tratados.

Após a apresentação dos principais procedimentos semânticos só falta mostrar como funcionam as rotinas de busca por métodos e propriedades de objetos. Quando se busca por métodos ou propriedades de objetos toda a informação que se tem é o nome da variável e o nome da propriedade ou método com seus parâmetros, caso existam. O que é feito neste caso é uma busca inicial pelo tipo da variável atual. Como esta variável não é de um tipo padrão Java, ela possui o campo tipo_aux diferente de nulo na estrutura de dados, e, a partir deste ponteiro chega-se diretamente ao objeto original. Depois dentro deste objeto basta pesquisar pelo método ou propriedade em questão. Caso a pesquisa em questão seja por um método, utiliza-se também o tipo dos argumentos e a quantidade de argumentos como critérios de pesquisa, uma vez que é permitido ocorrer *overloading* em Java. Independente se é uma busca por um método ou por uma propriedade, caso não seja encontrado na classe a propriedade ou método que se está procurando, vai se subindo na hierarquia, ou seja, começa uma busca recursiva pelos ancestrais desta classe até que seja encontrada a propriedade ou método informado. Caso ela não seja encontrada, uma mensagem de erro é retornada. Depois de encontrada a propriedade ou método em questão, é feita uma verificação dos seus modificadores e a partir deles, é definido se é permitido ou não chamar este método ou propriedade de dentro daquela classe.

A única característica da linguagem Java que não foi abordada pelo módulo foi a funcionalidade *interface*. O funcionamento desta funcionalidade é relacionado a parâmetros externos ao contexto da hierarquia de classes, e, por isso, optamos por deixá-la de fora do processo de engenharia reversa.

Como uma única observação final acerca do módulo que foi implementado, gostariamos de salientar que, todos os erros, com exceção da criação de classes herdeiras de classes finais e a chamada de classes ou métodos abstratos, resultam em erros do tipo warning durante a engenharia reversa. Isto foi definido desta forma porque os outros erros não implicam em praticamente nenhuma alteração da hierarquia de classes gerada. Por exemplo, se uma variável do tipo inteiro recebe um valor decimal, somente um warning é gerado. É claro que este erro é grave, e no caso de se tentar compilar este programa um erro fatal será retornado pelo compilador utilizado, porém para o contexto de hierarquia de classes o fato desta atribuição ser inválida não acarreta nenhum erro.

4.3.3 Rotinas de Transformação da Estrutura Interna para a Estrutura do Class Design

Após o término da fase de análise sintática do arquivo desejado, dá-se início à fase final do processo de engenharia reversa. O processo de carga da estrutura de dados do programa *Class Design*. Além desta carga, alguns procedimentos internos ao programa também tiveram que ser chamados de modo que a nova hierarquia de classes fosse mostrada ao usuário.

As rotinas de transformação da estrutura interna do módulo para a estrutura do programa são bastantes triviais, pois, a estrutura do programa é somente um subconjunto da estrutura interna do módulo. Todas as rotinas que foram criadas visavam somente fazer transformação de inteiro para *string*, com os modificadores, tipos de dados, mas, ainda, foi necessário a criação de um procedimento que gerasse a assinatura da classe, propriedade ou método. Sendo que a assinatura, nada mais é do que os modificadores seguidos pelo tipo, seguido pelo nome, seguido pelos parâmetros, caso existam.

A estrutura de dados do programa *Class Design* é dividida em dois objetos, CNodo e CMember, sendo que o primeiro armazena todas as informações acerca das classes e o segundo armazena todas as informações acerca dos métodos e propriedades.

A tarefa então foi, seguir a tabela de símbolos e criar as classes correspondentes, e, dentro de cada classe, criar as propriedades correspondentes, caso existissem. Quando toda a tabela de símbolos estiver mapeada, basta que se retorne o apontador para a classe topo e em seguida chamar a rotina de atualização(*redraw*) da tela do programa.

Com isso, concluímos a descrição da implementação do programa. Gostaríamos de ressaltar que existem muitas outras decisões de implementação que não foram sequer citadas aqui, pois somente discutimos aquelas que consideramos relevantes para a qualidade e funcionalidade do módulo.

5 Metodologia de Implementação da Hierarquia de Classes Java

5.1 Introdução

Esta seção apresenta as estruturas de dados usadas para a implementação da nova versão da ferramenta `Class Designer`[2], bem como as decisões de projeto.

A ferramenta Class Designer tem como propósito fundamental o auxílio ao programador nas fases iniciais do desenvolvimento de um sistema, a análise e o projeto. Ela permite ao usuário a geração de código para a linguagem alvo do sistema a medida em que ele vai sendo projetado. Além disso, caso tenha que ser feita mudanças posteriores, os arquivos fontes são atualizados para refletir estas mudanças. Esta facilidade é de vital importância a qualquer desenvolvedor pois economiza tempo e esforço. Muitas vezes um sistema tem de ser refeito devido a um erro durante a fase de projeto.

Class Designer foi inicialmente desenvolvida, para gerar código fonte para a linguagem Ita[1]. Nesta nova versão, os arquivos foram gerados para a linguagem Java[3] respeitando a hierarquia das classes do projeto, bem como seus membros definidos, propriedades e métodos.

Class Designer permite também que sejam adicionados a uma classe, os cabeçalhos de seus membros além do cabeçalho da própria classe em questão. Como consequência é necessário realizar uma análise a fim de impedir que um cabeçalho errado possa ser inserido pelo usuário. Também foi desenvolvido nesta fase do projeto um analisador sintático que analisa o que foi digitado pelo usuário a fim de verificar a validade daquele cabeçalho.
A Figura 1 apresenta a tela principal de ClassDesign-Java.

Figura 1: Tela Principal do Class Designer

5.2 Ambiente de Desenvolvimento

`Class Designer`[2] foi desenvolvida em ambiente Windows utilizando o ambiente de programação Visual C++ 1.0. A utilização deste ambiente se deu pelo fato da primeira versão do `Class Designer`[2] ter sido desenvolvida nele. O custo para a construção de uma nova

ferramenta partindo de outra linguagem iria ser muito alto se tivéssemos escolhido outro ambiente, visto que diversos componentes do programa teriam que ser implementados novamente.

Todavia, ao utilizar este ambiente mais antigo, a ferramenta teve que se adequar as suas limitações. Uma dessas limitações se refere ao fato da linguagem só manipular arquivos com no máximo oito caracteres e terminação com três caracteres. O tamanho dos arquivos utilizados pelo programa também se restringiu a 64KB.

5.3 Estruturas de Dados do Class Designer

Como Ita[1, 6] apresenta algumas diferenças em relação a linguagem Java[3], algumas alterações tiveram de ser feitas no código fonte original para esta nova versão.
Entre as diferenças existentes, as que mais influenciaram o projeto da nova versão de `Class Designer`[2], foram:

- Java não tem asserções
- Java não tem arquivos de cabeçalho
- Java não tem classes friends
- Java não tem parâmetros genéricos

Essas características fizeram com que a estrutura de dados utilizada pelo `Class Designer`[2] fosse alterada. Ela apresenta uma estrutura de dados que guarda informações sobre todas as classes inseridas em um projeto.
Um projeto no `Class Designer`[2], consiste de um arquivo com a terminação .cdn. Ele é criado quando o usuário fornece o nome do seu projeto. Dentro desse arquivo fica armazenado o conteúdo das estruturas de dados.
As estruturas de dados consistem primordialmente de duas classes, CNodo e CMember.
A classe CNodo apresenta as seguintes propriedades:

CString name, // nome da classe
base, // classe base
file, // arquivo de implementacao
fFile; // caminho absoluto para arq. implementacao

BOOL abstract, // classe abstrata
uptodate, // atualizada
final; //classe final

int level; // nivel hierarquico
int numDeferred; // numero de metodos postergados
CString identEscopo; // Public,private,protected ou nenhum

CObList* memberList; // lista de membros
CStringList* proprietyList; // lista de propriedades
CRect* classFrame; // posicao da classe na tela

CNodo *left, //ponteiro para classes pai, filha e irma
right,
up;

Com estas propriedades toda a hierarquia definida pelo usuário em um projeto, bem como as informações adicionais sobre a classe ficam armazenadas.

Uma classe, que é representada por um objeto do tipo CNodo, possui :

- um nome;
- uma classe base ou pai localizada acima dela na hierarquia;
- o nome do seu arquivo onde ela está implementada;
- o caminho completo para se chegar a esse arquivo.

Além desses itens enumerados ela possui propriedades que nos informa se ela é ou não abstrata ou final, se ela esta atualizada, assim como o seu nível na hierarquia.

O número de métodos postergados(abstratos) presentes nela fica armazenado em numDeferred. A propriedade identEscopo trata de guardar qual é o identificador de escopo para aquela classe. Este identificador pode ser *public, private, protected* ou simplesmente nenhum deles.

A lista de membros de uma classe, seus métodos e propriedades ficam armazenados em um vetor de objetos do tipo CMembro que será analisado mais adiante. A lista de propriedades de uma classe e sua posição na tela também são informados. As propriedades que tratam de relacionar a classe atual com as outras são *left, right e up*. Elas são ponteiros para outros objetos do tipo CNodo, ou seja, outras classes. A hierarquia foi implementada como sendo uma lista encadeada onde cada nodo é uma classe. A classe apresenta um ponteiro para cima onde se encontra sua super classe ou classe pai, localizada logo acima dela na hierarquia. Apresenta também um ponteiro para a direita apontando para a classe irmã a ela e um ponteiro para esquerda apontando para a classe filha. Com isso toda a hierarquia de classes pode ser armazenada sem problema.
A classe CMember é utilizada para guardar informações sobre os membros de uma classe :

char member; // D - membro de dado F - membro funcao

CString name, // identificador
type, // valor de retorno
signature; // assinatura

CStringArray parList; // lista de parametros

BOOL abstract; // metodo abstrato

BOOL final; // membro final
BOOL statico; // membro statico

CString idents;
CString identEscopo; //identificador de escopo

Esta classe guarda toda a informação necessária sobre um membro de uma classe, como seu nome(name), se ele é um membro de dado(propriedade) ou função(método). A assinatura do membro consiste do seu nome seguido do seu tipo de retorno e o tipo de seus parâmetros. A lista desses parâmetros é armazenada em um vetor de strings. O membro pode ser também abstrato, final ou estático. Da mesma forma que a classe, o seu membro irá ter um identificador de escopo, que pode ser *public,private,protected* ou nenhum deles. A propriedade idents trata de quardar todo o identificador armazenado antes do nome da classe. Ele representa na verdade, uma combinação das outras propriedades.

5.4 Geração de Class Designer Java

Para gerar Class Designer Java foram efetuadas varias alterações no código implementado para a linguagem Ita[1, 6]. Somente aqueles procedimentos que tratavam de operar diretamente na hierarquia é que não foram alterados. Porém apesar de algumas alterações a estrutura de trabalho dos arquivos das classes do `Class Designer`[2] se manteve a mesma. Esta estrutura se baseia em comentários inseridos nos arquivos das classes a fim de que se torne possível encontrar um membro ou mesmo a classe no seu arquivo por meio deste comentário.
As seguintes funções foram criadas :

BOOL ParametroValido(CString,CComboBox):

Recebe como parâmetro uma *string* que representa o parâmetro do membro em questão e a analisa a fim de verificar se ela é um parâmetro válido para Java[3]. O parâmetro é da forma TIPO IDENTIFICADOR, onde TIPO se refere ao tipo do parâmetro e IDENTIFICADOR ao nome dado ao parâmetro. Ele retorna TRUE se for válido e FALSE caso contrário. Diversas verificações são feitas nessa função, como por exemplo, se o nome do identificador não representa uma palavra reservada em Java[3], se o seu tipo , é um tipo válido, etc.

BOOL TipoValido(CString):

Recebe como parâmetro uma *string* que representa o tipo de retorno de uma função em Java[3] e verifica se o tipo é um tipo válido como retorno. Além de analisar se ele é um tipo básico, são percorridas também as classes da hierarquia a fim de verificar se alguns desses tipos estão relacionados a alguma classe em questão. Somente classes que são públicas podem ter seu tipo utilizados em outras.

BOOL IdentificadorValido(CString):

Recebe como parâmetro uma string que representa o identificador de uma função em Java[3] e verifica se ele é um identificador válido. O Identificador representa o nome do

membro(função ou dado). Ele é verificado a fim de analisar se ele já não existe para este membro, ou se ele não representa uma palavra reservada em Java[3].

6 Conclusão

Como a orientação por objetos [4] está fundamentada em objetos e classes, um aspecto essencial em qualquer metodologia orientada a objetos [4] é a determinação de classes e objetos que irão compor o modelo. De acordo com [2] uma das dificuldades encontradas durante o projeto de um sistema orientado por objetos [4] é a obtenção dos objetos e, ou classes e também a fatoração das propriedades comuns para a construção das várias hierarquias. Com isso inúmeras questões surgem, por exemplo, como determinar os objetos; como é feita a comunicação entre os vários objetos; que serviços oferecer; a que classe pertencem; como determinar os serviços de novas classes para reaproveitar os serviços das classes já existentes etc.

Com a criação da nova versão de *Class Design*, *ClassDesign-Java*, podemos, agora, utilizar todo o potencial de um projeto orientado por objetos tanto em um projeto novo, quanto em um projeto já existente. Isto é possível graças ao fato deste novo programa estar perfeitamente apto a funcionar tendo como linguagem alvo, Java.

Especificamente, a criação de uma hierarquia de classes a partir de um projeto em Java está concluída com sucesso. Todos os objetivos iniciais foram alcançados e agora o programa gerado possui a capacidade de fazer a engenharia reversa de um projeto, permitindo que seja possível se visualizar a sua hierarquia de classes.

Obviamente, ainda existem coisas a serem melhoradas dentro do programa. Uma delas seria a tradução do código fonte do programa para uma linguagem mais atual, que possa fazer uso da arquitetura de 32 bits e de mais do que 64K de memória. Infelizmente esta versão do programa possui estas limitações não devido à maneira como foi programado mas devido ao fato de ter sido feito em Visual C++ 1.0 (aplicativo que gera código em 16 bits).

Outro ponto que poderia ser melhorado dentro do programa, mais especificamente dentro da fase de engenharia reversa, é o fato de não haver nenhum tratamento do comando *interface*. Deve ser feito uma ampliação na representação gráfica de modo que seja possível mostrar as *interfaces* de uma classe sem que isto prejudique a vizualização dos ancestrais desta classe.

No geral podemos dizer que todos os objetivos do projeto foram alcançados com sucesso, e que pudemos aprender muito com ele. Principalmente de orientação por objetos e de compilação de programas orientados por objetos.

7 Bibliografia

Referências

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

[2] 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.

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

[4] Bertrand Meyer, Object-oriented Software Construction, Editor C. A. Hoare, 1998.

[5] http://www.Javasoft.com/docs/books/jls/html/19.doc.html

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

[7] Bertrand Meyer, From Structured Programming to Object-Oriented Design: the road to Eiffel,

Structured Programming, 1988.

[8] Hanspeter Mössenbock & Wirth, The Programming Language Oberon-2, Technical Report, 1992

[9] Lamas, José Raphael de Souza *Programa Class Design para Java*,

Relatório Final de POCII. Departamento de Ciência da Computação, ICEx, UFMG, Julho de 1999.

[10] Arthur Zambelli de Almeida,Geração da Definição de uma Hierarquia de Classes a partir de código Java,Relatório de Projeto Orientado em Computação, DCC-ICEX, UFMG, 1999.