Programação por Objetos:
Conceitos, Linguagens e
uma Experiência

Dilma Menezes da Silva

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE
MESTRE EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Arnaldo Mandel**

— São Paulo, Agosto de 1990 —

Abstract

In this work the ideas of object-oriented programming are analyzed from different points of view. Our objective is to evaluate the strength of these ideas in expressing abstractions and producing reusable software. At first, we try to detect at what stages a programming paradigm impacts the development of software. Then we study three possible ways for conceptualizing object-oriented programming and relate them to the original works that introduced object-oriented ideas. The third part of this dissertation focuses on the formal aspects of these concepts, using type theory. In chapter four we discuss the implementation of *objects*, *classes* and *inheritance* in some object-oriented programming languages. We finish our investigation discussing our experience in designing a library of routines for manipulating graphs. This work provided us with the opportunity of dealing with generic collections, parameterized types and multiple representations for abstract data types. We conclude that object-oriented programming offers appropriate principles for producing software but is far from being a solution for all system development problems.

Agradecimentos

Estou com vergonha de parecer *tão melada* e com receio de não conseguir manifestar o carinho que sinto pelos que me acompanharam nessa jornada, mas também com bastante vontade de deixar aqui alguns recados:

Arnaldo, trabalhar com você é uma delícia. Super obrigada por *tudo*!

Carlinhos, Yô, Mário, Dê, Nê e Claudia, sei que não faz sentido agradecer amizade, mas obrigada pela força!

Alex, Cris, Coelho, Edna, Steve, Sônia e Mané, obrigada pela ajuda com os cursos e pela companhia nas “férias”;

Edna, Adenilza e Joelma, vocês sempre tão prestativas e eu em geral tão mandona ... Desculpem-me!

Humberto, Denise e “meus” monitores da Poli, obrigada pela dedicação;

Aluninhos do BCC e da Poli, desculpem-me pelo semestre atropelado;

Routo e Paulo F., obrigada pelo apoio;

Valdemar e Nami, obrigada pela atenção e carinho desde a época da minha graduação;

Painho, Mainha, Delma e Claudio, a parte mais chata desse trabalho foi passar tanto tempo longe de vocês.

Conteúdo

Introdução	5
1 Software e Paradigmas	7
1.1 A perspectiva da engenharia de software	7
1.2 O papel da programação	9
1.3 Paradigmas de programação	10
1.4 Paradigmas e linguagens de programação	11
1.5 Paradigmas e formalização	12
1.6 O paradigma da programação por procedimentos	13
1.7 O paradigma da programação funcional	13
1.8 O paradigma da programação em lógica	14
1.9 Outros paradigmas	14
2 Programação por Objetos	16
2.1 A conceituação através de modelos físicos	18
2.1.1 Prós e Contras	20
2.1.2 A conceituação e linguagens	21
2.2 Conceituação através de objetos e mensagens	21
2.2.1 Prós e Contras	24
2.2.2 A conceituação e linguagens	26

2.3	Conceituação baseada em reutilização de software	27
2.3.1	Apresentação do paradigma	29
2.3.2	Prós e Contras	30
2.3.3	Linguagens	30
2.4	Discussão sobre as três apresentações	30
3	Conceitos e Formalização	32
3.1	Idéias básicas em programação por objetos	32
3.1.1	Tipos	32
3.1.2	Tipos Abstratos de dados	34
3.1.3	Encapsulamento	35
3.1.4	Classes e objetos	35
3.1.5	Herança	38
3.2	Apresentação formal dos conceitos	45
3.2.1	Teorias e tipos	45
3.2.2	Teorias e encapsulamento	54
3.2.3	Teorias e herança	55
3.2.4	Herança em OBJ2	59
3.2.5	A especificação para herança em GSBL	60
3.2.6	Tipos, de novo	61
3.3	Polimorfismo	64
3.4	Herança múltipla	67

4	Linguagens e Objetos	69
4.1	Linguagens	69
4.1.1	C++	72
4.1.2	Smalltalk	81
4.1.3	Outras Linguagens	87
4.2	Algoritmos para herança múltipla	94
5	Uma Experiência com Programação por Objetos	98
5.1	O projeto: uma biblioteca para grafos	99
5.1.1	Motivação	100
5.1.2	Primeiro contato: análise informal de requisitos do software . .	101
5.1.3	Objetos têm nomes?	106
5.1.4	A questão da estrutura de dados	106
5.1.5	Implementando uma lista genérica	107
5.1.6	Implementando tipos parametrizados	112
5.1.7	Reinventando a roda	112
5.1.8	Tratando as alternativas para armazenamento de um objeto . .	115
5.1.9	Múltiplas representações para um mesmo objeto	118
5.2	Considerações finais sobre o projeto	120
5.2.1	Estágio atual do software	122
5.2.2	O projeto como aprendizado	123
5.3	Em busca de uma metodologia	125
5.4	Afinal, o que é programação por objetos?	126
5.5	Outras questões interessantes	127

A	A especificação de algumas classes	128
B	Alguns exemplos	136
	Bibliografia	139
	Índice	147

Introdução

Programação por objetos é um conjunto de idéias sobre desenvolvimento de programas que há algum tempo vem causando sensação e que acabou se tornando a estrela do momento no cenário de desenvolvimento de sistemas. Uma parte de sua legião de admiradores enxerga em seus princípios uma revolução, uma ruptura com a forma tradicional de desenvolvimento de programas. Há também os *fãs* menos empolgados, para os quais programação por objetos significa uma evolução da linha atual de construção de sistemas que atenua alguns dos problemas da produção de software. Como em toda carreira de sucesso, não há como destacar uma razão única para a notoriedade, mas certamente programação por objetos conquista muito espaço ao acenar com a possibilidade de reutilização do software já desenvolvido.

O paradigma de programação por objetos conseguiu atizar a curiosidade de boa parte da comunidade envolvida com desenvolvimento de programas. Podemos observar suas idéias avançando em várias frentes através da enorme quantidade de artigos *com objetos* publicados; ao mesmo tempo em que linguagens de programação orientadas a objetos são projetadas vários trabalhos com semântica denotacional e teoria dos tipos lidam com os conceitos do paradigma. É inevitável que nos questionemos: o que há por trás dessa estória toda? O que é programação por objetos, a que veio, para onde vai?

Embora vários aspectos do paradigma de programação por objetos tenham nos fascinado desde o primeiro contato com o assunto, nosso trabalho é uma tentativa de explorar o mundo dos objetos sob a visão de um alienígena, ou seja, não tivemos pressa em adquirir a cultura associada ao desenvolvimento orientado por objetos. Como resultado esperamos apresentar uma visão crítica para a utilização dos *objetos* que saliente os vários aspectos positivos dessa forma de desenvolvimento.

Nossa investigação começa no capítulo um, onde sobrevoamos as idéias de desenvolvimento de software a fim de discernir quais estágios da atividade são diretamente atingidos pela escolha de um paradigma de programação. A intenção do capítulo dois é coletar “o que dizem por aí” acerca do paradigma de programação por objetos; para isso estudamos os conceitos e suas origens através das diversas apresentações encontradas na literatura. No capítulo três tentamos esquecer tudo que ouvimos de outrém e partimos em busca de motivações para cada um dos conceitos. Para isso revisitamos noções usuais em programação como *tipos* e *encapsulamento* e procuramos introduzir os novos conceitos — como *classe* e *herança* — com base no poder de abstração que acrescentariam ao software. Após conceituarmos — de maneira informal e desvinculada de linguagens — cada uma das idéias básicas apresentamos várias abordagens formais e linguagens para especificação algébrica associadas a esses conceitos. No capítulo quatro são focalizados os pontos que julgamos mais interessantes na implementação dos conceitos de programação por objetos em uma linguagem de programação: estruturas para armazenamento de classes e tratamento de herança (inclusive herança múltipla). No capítulo cinco relatamos a experiência em desenvolver uma biblioteca de rotinas para manipulação de grafos, discutindo como os conceitos estudados atuam na modelagem. O trabalho de programação dessa biblioteca deu margem a questões muito interessantes relacionadas a implementação de polimorfismo, coleções genéricas, tipos parametrizados e gerenciamento de estruturas de dados alternativas para um tipo abstrato de dados. Encerramos nosso trabalho tecendo algumas conclusões sobre o paradigma de programação por objetos como metodologia de desenvolvimento de software e sua eficácia em prover reutilização de software.

Capítulo 1

Software e Paradigmas

O objetivo principal deste capítulo é investigar a localização de um *paradigma de programação* na atividade de desenvolvimento de software. Em nossa análise preferimos utilizar as idéias de *engenharia de software*, que fornecem uma divisão da atividade de produção conveniente para situar *programação* no processo de desenvolvimento de sistemas.

1.1 A perspectiva da engenharia de software

O desenvolvimento de software foi considerado por algum tempo como uma atividade essencialmente artesanal. Os erros de programação e problemas presentes no produto final eram considerados temporários e atribuídos à falta de experiência com as novas máquinas. Décadas de trabalho não eliminaram os problemas iniciais; ao contrário, os mesmos foram acentuados com o passar do tempo e a necessidade de sistemas cada vez mais complexos. Além dos problemas com a qualidade do software produzido, questões como alto custo e dificuldade para cumprimento de prazos estabelecidos tornaram-se parte integrante do cenário de desenvolvimento de sistemas.

Em resposta a essa situação — comumente chamada de *software crisis* ([Pr 82], [HM 84]) — surgiu a **engenharia de software**, que pode ser encarada como um conjunto de disciplinas e ferramentas empregadas a fim de tratar a complexidade do software, aumentar a confiabilidade e corretude dos sistemas e desenvolver técnicas para previsão de custos. Em [Hum 88] podemos encontrar as seguintes definições:

- *software*: O termo software se refere a programas e todas as informações e materiais necessários para sua instalação, operação e manutenção.
- *engenharia de software*: aplicação disciplinada de princípios e métodos

matemáticos e científicos na produção de software de boa qualidade.

A engenharia de software enfrentou como desafio inicial a identificação dos principais conceitos e componentes do processo de desenvolvimento de sistemas. A pesquisa evoluiu de maneira que o desenvolvimento de software passasse a ser analisado através de sua divisão em estágios, facilitando o controle do projeto e a avaliação do produto final. Podemos destacar duas idéias básicas que fundamentam a engenharia de software:

1. Ciclo de vida do desenvolvimento de software.

Existem várias propostas para uma divisão das atividades que compõem o desenvolvimento. A mais usual é formada pelas seguintes fases ([ZSG 79], [Sa 80]):

- (a) análise dos requisitos do sistema
São levantados os objetivos do projeto em termos de funcionalidade, operacionalidade, eficiência, custos, tempo de desenvolvimento.
- (b) especificação do sistema ([MA 88])
Define formalmente quais operações serão implementadas, obtendo uma descrição detalhada do aspecto funcional do sistema.
- (c) projeto
Esta fase visa organizar a implementação (programação) das especificações do sistema. Divide-se o sistema em partes (módulos), cada qual com uma função bem definida. São escolhidos métodos (algoritmos) para desenvolver cada módulo. Alguns pesquisadores consideram a fase de projeto como a fase central e unificadora da atividade de software; muita pesquisa tem sido feita ([Fr 80]) nessa área abrangendo gerenciamento do projeto, metodologias, técnicas e ferramentas apropriadas.
- (d) codificação
Fase onde se obtém código para cada módulo do sistema. É considerada uma fase “fácil”, no sentido em que é a mais bem conhecida. A maior parte dos problemas que ocorrem nessa fase são devidos a falhas na fase de projeto.
- (e) testes (depuração)
Cerca de cinquenta por cento do esforço na produção de um sistema novo se concentra nesse estágio. A fase de testes deve ser encarada como o “processo de executar um programa com a intenção de encontrar erros” ([My 79]).
- (f) manutenção
É a atividade que ocupa a maior parte do ciclo de vida de um sistema e se refere a correção de erros encontrados e adaptações do sistema provenientes de exigências não presentes na primeira fase do desenvolvimento.

Embora o conceito de ciclo de vida esteja presente em grande parte dos trabalhos em engenharia de software, há pesquisadores ([Gl 82]) que acreditam que esse

conceito deve ser abolido. Há propostas de eliminar a idéia de “requisitos do sistema” trocando-a por “objetivos do sistema” e propostas de novas divisões entre as fases de “requisitos” e “operação e manutenção” ([Bl 82]).

Não há garantias de que uma visão tradicional para o desenvolvimento de sistemas se adeqüe à divisão de atividades envolvida na produção de software através *programação por objetos*. Ao contrário, muitos trabalhos tentam definir um ciclo de vida especial para essa forma de desenvolvimento de sistemas. Nós insistimos em analisar a produção de software através das idéias tradicionais de ciclo de vida a fim de buscar através de uma forma usual de enxergar a produção de sistemas a área de atuação de um paradigma de programação.

2. Software de Boa Qualidade.

Foi necessário substituir a idéia “pragmática” de que um bom programa é um programa que funciona. O conceito de software de boa qualidade passou a englobar aspectos como:

- obtenção de software cuja rapidez, eficiência e funcionalidade venham de encontro aos requisitos do sistema;
- facilidade de manutenção e expansão do sistema;
- documentação presente no código e documentação descrevendo o desenvolvimento do sistema.

Foram criadas medidas (*software metrics*) para qualidade e complexidade de software ([Bas 80]).

1.2 O papel da programação

Chamaremos de *programação* o conjunto das atividades desenvolvidas durante as fases de projeto e codificação. É através da programação que obtemos o código correspondente às especificações.

A importância da *fase* de programação se deve principalmente às decisões que são tomadas nesse estágio: o caminho seguido determinará características importantes do restante do desenvolvimento do sistema.

A fim de evitar falhas “congenitas” no projeto é necessário um discernimento correto de *quais* são as escolhas necessárias para que se dê início à atividade de programação. Antes da análise de questões como equipamento, linguagem de programação, ambiente, metodologia e forma de documentação se deve adotar uma *filosofia de programação conveniente à particular aplicação*. Em outras palavras, a primeira decisão importante no processo é a adoção de um **paradigma de programação**.

1.3 Paradigmas de programação

Podemos encontrar no dicionário Aurélio [Au 86] a seguinte definição para o vocábulo *paradigma*:

[Do grego *paradeigma*, pelo latim *paradigma*.]

1. Modelo, padrão, estalão
2. (Gramática) Modelo ou tipo de conjugação ou declinação gramatical.

Em *Random House Dictionary* encontramos “an example serving as a model”.

Vários autores utilizam a palavra *paradigma* como conceito usual na área de computação. Não encontramos um trabalho que entre em detalhes quanto ao significado da expressão “paradigma de computação”. Madsen [K-M 88] prefere utilizar em seus escritos o termo *perspectiva* em lugar de *paradigma*. Nygaard [Ny 86] escreveu um artigo [N-S 87] onde questiona o uso de “paradigm”; infelizmente não tivemos acesso a sua argumentação.

Após estudar os diversos paradigmas existentes concluímos que em *paradigma de programação* pode ser visto como um estilo de programar. Os diversos estilos diferem fundamentalmente no que se refere a:

- modelo de máquina, ou seja, como é vista a transformação de uma instância de entrada em sua solução correspondente;
- análise de dados: quais aspectos do domínio da aplicação são relevantes ao processamento ou enfatizados na programação;
- disciplina : quais princípios regem a divisão do programa em partes de menor complexidade.

Veremos de maneira sucinta vários paradigmas e estudaremos em detalhe o *paradigma da programação por objetos*.

1.4 Paradigmas e linguagens de programação

O papel da linguagem de programação no processo de desenvolvimento de software deve ser analisado levando em consideração as diferentes áreas em que ela atua [K-M 88]:

1. Modelagem Conceitual

A linguagem de programação é uma maneira de expressar conceitos, fenômenos e estruturas do domínio da aplicação;

2. Instrução do computador

Através da linguagem de programação são focalizados aspectos da execução do programa como armazenamento e fluxo de informação;

3. Gerenciamento da descrição do programa

As linguagens de programação apresentam características que determinam o aspecto geral do programa em termos de modularidade, acesso a variáveis, uso de bibliotecas, etc.

As diferenças entre as diversas linguagens de programação podem ser detectadas na maneira como trata cada uma das três atividades acima. Encontramos maneiras distintas de expressar a informação (1), diferentes mecanismos de armazenamento, manipulação e controle de fluxo (2) e diferentes enfoques para a concepção geral do programa (3). A comparação das linguagens não pode ser restrita ao simples confronto entre mecanismos e facilidades oferecidas. Uma linguagem não é melhor que outra por possuir um determinado mecanismo; muito menos é marcante a quantidade de facilidades oferecidas. É importante que:

- os mecanismos fornecidos estejam integrados na linguagem de maneira clara e elegante;
- seja possível combinar os mecanismos a fim de evitar a introdução de mecanismos novos voltados para casos especiais;
- a implementação de um mecanismo da linguagem não cause custos adicionais (tempo, espaço) em programas que não o utilizem.

Um enfoque quantitativo eliminaria da análise fatores importantes como simplicidade, consistência e ortogonalidade. A escolha da linguagem de programação a ser utilizada no desenvolvimento de um sistema deve ser guiada pelo estilo de programação adotado, deve ser feita optando-se pela linguagem que melhor *forneça suporte* ao paradigma de utilizado (que por sua vez, foi escolhido em função da particular aplicação a que se destina o sistema).

Dizemos que uma linguagem de programação fornece suporte [Str87a] a um paradigma de programação se a linguagem dispõe de mecanismos que tornam o uso do paradigma razoavelmente fácil, seguro e eficiente. A linguagem não fornece suporte a um paradigma quando requer esforço ou habilidades excepcionais para a utilização do mesmo. Por exemplo, podemos escrever programas estruturados em Fortran ou programas com verificação de tipos em C, mas um certo esforço extra seria despendido uma vez que essas linguagens não fornecem suporte ao estilo adotado.

O suporte para um paradigma não deve vir somente através dos mecanismos (“comandos”) da linguagem que permitem o uso do paradigma. São importantes verificação em tempo de compilação e execução que visem detectar desvios não intencionais do paradigma, ambientes de programação e bibliotecas apropriados.

Por último, gostaríamos de diferenciar suporte de capacidade de simulação através de um exemplo envolvendo os comandos *repeat* e *while*. Sabemos que a partir de um desses comandos podemos simular o outro; assim o conceito de repetição expresso pelo *repeat* pode ser utilizado em uma linguagem que possua apenas o comando *while*. Tal linguagem simula o conceito do *repeat*, mas não fornece suporte para esse conceito, já que toda vez que desejarmos utilizá-lo deveremos recorrer ao artifício de simular o *repeat* a partir do *while*.

É necessário cuidado para não cair no erro de comparar o poder de expressão de duas linguagens através da capacidade de simular com uma os comandos da outra.

1.5 Paradigmas e formalização

“What can be said at all, can be said clearly; and on what we can not talk about, we have to remain silent”

Wittgenstein

Antes de introduzir os diversos paradigmas é interessante destacar a necessidade da existência de uma boa formalização e conceituação para os paradigmas.

Se acompanharmos o andamento histórico da pesquisa em programação observaremos que as idéias iniciais eram confusas ou apelavam para explicações em termos de linguagens de programação. Alguns textos pareciam supor um envolvimento prévio do leitor com as novas idéias. Só com o amadurecimento dessas idéias surgiam boas conceituações e trabalhos que *ensinavam* um novo estilo ou conceito.

Mas como formalizar as idéias presentes em software? Dijkstra levanta em [Dijk] uma excelente discussão acerca do que deve ser considerado um trabalho bem escrito na área de programação. Ele diz que o texto deve ser “matemático”¹, sem que isso

¹no sentido dado por Moris Kline: “mais do que qualquer coisa, matemática é um método”

signifique o uso do estilo de escrever tradicional da matemática.

Observe que alguns dos paradigmas que apresentamos a seguir estão ancorados em uma boa formalização matemática, sendo suas caracterizações mais simples e elegantes.

1.6 O paradigma da programação por procedimentos

Também chamado de programação procedimental ou imperativa [CGF 86], o paradigma da programação por procedimentos foi o primeiro a aparecer e espelha em alguns de seus conceitos básicos o modelo de computação de von Neumann (como por exemplo na atribuição, execução condicional, execução seqüencial). A esse paradigma foi dedicada grande parte da pesquisa em programação e ainda hoje é o mais utilizado no desenvolvimento de sistemas.

Sob o ponto de vista desse paradigma a execução de um programa é tida como uma seqüência de chamadas de procedimentos que manipulam estrutura de dados. O enfoque central é descrever *como* efetuar o processamento necessário à computação desejada. A utilização de procedimentos organiza essa descrição dividindo o processamento em partes menos complexas.

Uma linguagem de programação que forneça suporte para esse paradigma deve apresentar mecanismos que implementem as idéias de funções, procedimentos e parâmetros. O conceito de atribuição também deve estar presente em uma tal linguagem, já que a idéia de *estado da memória* — configuração das células de armazenamento — desempenha um papel importante na descrição dos passos seguidos durante o processamento.

Algumas linguagens que fornecem suporte a esse paradigma são: Fortran, Algol 60, Algol 68, C e Pascal.

1.7 O paradigma da programação funcional

Sob a perspectiva da programação funcional um programa é uma função matemática que descreve uma relação entre a entrada e a saída. O conceito de *variável* ou *estado de memória* é eliminado, toda a computação sendo feita através da aplicação de funções. Alguns aspectos referentes a pesquisa nesse paradigma merecem destaque:

1. a tentativa de escapar da arquitetura von Neumann;
2. a especificação de soluções sem a preocupação com detalhes de implementação;
3. a possibilidade de se obter linguagens cuja semântica possa ser definida de maneira precisa, já que seus mecanismos básicos se adequam ao tratamento matemático.

Backus, em [Ba 78], desenvolveu uma descrição formal do que denomina *sistemas de programação funcional (FP systems)* e destacou vantagens referentes a (1) e (2).

Para que uma linguagem forneça suporte à programação funcional deve oferecer mecanismos para definição de funções e aplicações destas a argumentos. Nesse sentido, Lisp fornece suporte ao paradigma, embora textos em programação funcional [Me 88] frisem que essa linguagem não é funcional, já que muitos de seus “dialetos” apresentam facilidades para atribuição e controle.

Outras linguagens são Scheme [Schem] e Miranda [Tu 78].

1.8 O paradigma da programação em lógica

No processo de resolver um problema sob o paradigma da programação em lógica busca-se especificar as propriedades importantes do problema, ao invés de descrever uma possível solução. A especificação é feita através de um conjunto de sentenças, expressas em uma linguagem simbólica que elimine as ambigüidades e imprecisões inerentes ao uso de uma linguagem natural. Técnicas desenvolvidas a partir de mecanismos da lógica permitem enxergar o formalismo obtido como um programa executável.

A partir de 1973, com o desenvolvimento da linguagem Prolog (PROgramming in LOGic) e, um pouco depois, com os “sistemas especialistas” vários pesquisadores vêm trabalhando com o paradigma.

1.9 Outros paradigmas

Alguns autores consideram ainda a existência dos paradigmas de programação por processos e de programação por eventos [K-M 88]. Consideramos esses paradigmas como ramificações do paradigma de programação por procedimentos, já que o enfoque central ainda é descrever o processamento necessário pra a obtenção de uma solução.

Quanto aos paradigmas de encapsulamento (*data hiding*) e tipos abstratos de dados, preferimos enfocá-los não como estilos de programação em si, mas como integrantes de um processo de transição das idéias do paradigma da programação por

procedimentos em direção no paradigma de programação por objetos. Acreditamos que essa transição foi marcada por uma importante mudança de enfoque: da *ação* (descrição do modo de obtenção de uma solução) para os *dados* (estrutura da informação no domínio da aplicação). Essa mudança foi motivada pela ineficácia da programação por procedimentos em gerar software de boa qualidade para problemas grandes.

Estudaremos encapsulamento e tipos abstratos de dados no capítulo 3 como conceitos básicos para o acompanhamento de nossa abordagem para programação por objetos.

Capítulo 2

Programação por Objetos

"What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

Tim Rentsch, 1982

Existem várias linhas de conceituação para programação orientada por objetos. Os vários enfoques não são disjuntos em termos conceituais, mas abordam o paradigma com diferentes considerações históricas e motivações. As diferentes linhas não induzem apenas formas distintas para compreensão das idéias em programação por objetos, mas também destacam diferentes aplicações para as idéias apresentadas.

Os objetivos da nossa apresentação do paradigma são:

- analisar separadamente três conceituações existentes.

Muitos trabalhos apresentaram os conceitos básicos através de construções em linguagens de programação específicas. Além disso, usavam termos como *objetos*, *classes*, *herança* e métodos como se já fossem conhecidos e compreendidos pelo leitor. Nós tentamos, como outros trabalhos encontrados na literatura, desvincular linguagem dos conceitos e explicar qual o sentido associado a algumas "expressões primitivas" usualmente utilizadas.

Esperamos que ao final da apresentação das três conceituações o leitor tenha uma idéia geral (inicial) do que é programação por objetos que sirva de motivação para uma apresentação mais minuciosa de cada um dos conceitos que consideramos centrais ao paradigma (expostos no capítulo 3).

- analisar a relação entre as três conceituações, examinando a possibilidade de unificá-las — afinal, elas expressam o mesmo paradigma ou não?

Nós classificamos as apresentações do paradigma presentes na literatura em dois ramos: derivados de conceitos da linguagem SIMULA e derivados de conceitos de *tipos abstratos de dados* e *reutilização de software*. Esta divisão foi baseada principalmente na maneira de introduzir o paradigma no processo geral de produção de software. A classificação pode também ser considerada válida quanto à maneira de exposição dos conceitos básicos, mas nesse caso a divisão não é muito rígida.

Pequeno histórico da linguagem Simula

Na década de quarenta a simulação digital foi introduzida como uma importante ferramenta para a análise de situações onde nenhum modelo matemático parecia apropriado. Um dos primeiros usos da simulação foram o estudo de partículas atômicas utilizadas em reatores nucleares e análise de situações complexas de pesquisa operacional.

Muitas linguagens de simulação começaram a aparecer, cada uma representando uma visão padronizada dos sistemas a serem simulados [Ny 86]. Os primeiros exemplos foram Simscript, GPSS, CSL, Sol e Simula I.

Simula I foi projetada entre 1962 e 1964, estando disponível em 1965. A organização do processo de computação era feita através de um sistema formado pela execução de vários programas que interagiam. Este enfoque parecia ser útil a várias aplicações, donde Simula passou a ser usada como uma linguagem de programação de uso geral. Os componentes básicos da linguagem Simula I eram chamadas *atividades* e *processos*; essas idéias evoluíram e foi definida a linguagem Simula67 [D-N 66], primeira a associar sintaticamente dados e procedimentos.

Simula67 foi derivada da linguagem Algol60, como podemos perceber pelas suas características gerais de sintaxe. A principal diferença é quanto às estruturas de bloco e regras de escopo do Algol60, intencionalmente construídas para garantir que o bloco mais recentemente ativado fosse terminado primeiro. Para simulação eram necessárias entidades que possuísem períodos de ativação próprios da entidade e pudessem interagir uma com as outras. Em Simula67 estas entidades independentes são chamadas classes. A simulação pode envolver atividades paralelas entre as diversas entidades; esse paralelismo é tratado através do uso de *co-rotinas*.

Uma declaração de classes tem o seguinte formato:

```
[classe prefixo] class nome [(parâmetros)];  
    especificação dos parâmetros  
  
begin  
    declarações  
    comandos  
end
```

O trecho *begin-end* é uma espécie de bloco do Algol60, que pode conter procedimentos e declarações de outras classes. Os *parâmetros* são utilizados para a associação de valores iniciais a novos objetos de uma classe; a *classe prefixo* é utilizada para declarar uma classe a partir de outra já existente. Para dar nome a objetos de uma classe e referenciá-los são usadas *variáveis de referência* (pointers).

Um exemplo bem simples e ilustrativo de um programa escrito em Simula pode ser encontrado em [Ho 83].

2.1 A conceituação através de modelos físicos

Vários pesquisadores da Escandinávia vêm se destacando na pesquisa em programação por objetos; seus trabalhos abrangem metodologias, conceitos e linguagens. Nygaard, Madsen, Møller-Pedersen e Knudsen sugerem uma apresentação formal para o paradigma não necessariamente baseada em conceitos matemáticos como as conceituações existentes para programação em lógica ou funcional. Eles propõem que a ciência da computação seja encarada sob o ponto de vista mais geral, pois ao lidar com informação deixa de ser uma ciência abstrata como a matemática e passa a englobar o estudo de fenômenos do mundo real. Eles chegaram à seguinte definição [M-P 88]:

Sob o paradigma de programação por objetos a execução de um programa é tida como um **modelo físico** que simula o comportamento de uma parte real ou imaginária do mundo.

Para compreendermos esse enfoque do paradigma veremos a definição de *modelo físico* usada por esses pesquisadores, que é construída a partir de **fenômenos** e **conceitos**.

- *Fenômenos*

São chamados de fenômenos os componentes relevantes do processo que queremos modelar. O modelo físico deve retratar três aspectos do fenômeno: substância, propriedades mensuráveis da substância e transformações da substância. Vejamos o que são cada um desses aspectos através de um exemplo (em [M-P 88] a explicação também é feita através de exemplos; é adotada a personagem de quadrinhos Garfield como *fenômeno* e seus *aspectos* exemplificados através de “tiras”. Nosso exemplo, embora menos “interessante”, tenta explorar a modelagem de um mundo sem propriedades “físicas” usuais como peso, altura, etc). Suponhamos que queremos modelar o mundo da Teoria dos Grafos. Depararemos com grafos específicos que devem estar presentes no modelo já que desempenham papel importante na teoria, como o grafo de Petersen, o K_5 ou o $K_{3,3}$. Cada um desses grafos têm substância, no sentido de que podem ser individualizados e diferenciados do resto do modelo. Eles têm propriedades mensuráveis

(número de vértices, grau máximo, número de componentes, etc) e devem estar presentes nos modelos as ações correspondentes a várias transformações dessas propriedades (a ação de contração de uma aresta pode alterar o número de vértices, a eliminação de uma aresta pode tornar o grafo desconexo, etc)

- *Conceitos*

Muitas vezes a análise descritiva dos fenômenos não é a melhor maneira de obter uma boa modelagem. Para que possam ser captadas propriedades essenciais de um fenômeno é necessário desenvolver *abstrações* ou *conceitos*. A clássica noção de conceito envolve três elementos: *nome* (denotará o conceito), *intencionalidade* (quais são as propriedades do fenômeno cobertas pelo conceito) e *extensionalidade* (quais os fenômenos cobertos pelo conceito).

Conceitos são criados por abstração, buscando-se propriedades similares dos fenômenos e descartando-se diferenças. São três os mecanismos de abstração mais importantes:

1. classificação: mapeia os fenômenos em conceitos. A operação reversa é a exemplificação.
2. agregação: um novo conceito é abstraído juntando-se conceitos já existentes. Seu reverso é a decomposição.
3. generalização: um conceito é criado para generalizar conceitos já existentes. A operação reversa é chamada de especialização.

No nosso exemplo o conceito de *grafo* engloba os fenômenos grafo de Petersen, K_5 e $K_{3,3}$, entre outros. O conceito de grafo pode ser especializado em *grafos completos*, dos quais K_5 e $K_{3,3}$ são exemplos. O conceito de grafo também pode ser visto como derivado da agregação dos conceitos de conjunto e função de incidência.

Agora podemos ver a definição de modelo físico dada em [M-P 88]:

Um modelo físico consiste de *objetos*, cada objeto caracterizado por *atributos* e uma seqüência de *ações*. Objetos organizam o aspecto de substância dos fenômenos e transformações da substância são obtidas através da execução de ações pelos objetos. Objetos podem ter outros objetos como parte. Um atributo pode ser uma referência para um objeto; alguns atributos representam as propriedades mensuráveis do objeto.

O *estado* de um objeto é expresso através de uma substância, suas propriedades mensuráveis e as ações agindo sobre elas. O estado do modelo é a união dos estados de cada objeto presente no modelo.

Os elementos do modelo físico que representam conceitos são chamados *padrões*. Um padrão define as propriedades comuns de uma categoria de objetos. Padrões podem ser organizados em uma hierarquia. Padrões podem ser atributos de objetos.

As ações em um modelo físico são executadas pelos objetos. Uma seqüência de ações pode ser executada:

- *concorrentemente* a outras seqüências
- *alternadamente* (uma ação a cada vez)
- como parte das ações de um outro objeto

2.1.1 Prós e Contras

O enfoque de programação por objeto através de modelos físicos denota a preocupação em obter um programa que espelhe seu domínio de aplicação. A utilização dessa filosofia durante a fase de programação de um sistema, ao invés da programação visando atender exigências funcionais imediatas, nos conduz a alguns resultados favoráveis:

- obtenção de um software estável onde alterações e extensões se encaixam naturalmente. Supondo que o “mundo” modelado não tenha sofrido grandes alterações é pouco provável que a implementação de novos requisitos do sistema entre em conflito com o software já desenvolvido;
- o software resultante é mais compreensível, já que está mais perto da concepção que a equipe de desenvolvimento tem do “mundo”.

Enxergamos algumas possíveis críticas a essa filosofia de programação:

- nem todas as aplicações são conhecidas o suficiente para que sejam identificados os fenômenos e conceitos importantes;
- A manutenção de uma visão purista de modelagem pode levar a sistemas ineficientes; a inclusão de aspectos que levem a um processamento mais eficiente podem tornar o software artificial demais para ser considerado um modelo físico.

Por último, podemos fazer uma especulação filosófica. A apresentação pode ser criticada em função de sua origem, ou seja, do ponto de vista de seus idealizadores acerca de *ciência de computação*. Ao tratar computação como uma ciência relacionada ao estudo de fenômenos do mundo real há a possibilidade de entrarem em choque com pesquisadores que acreditam ser esta ciência, em última análise, puramente abstrata. Por exemplo, Weizenbaum conclui em [We 76] que existem questões que não podem ser descritas em termos de um *procedimento efetivo*¹ e Setzer em [Set 84] des-

¹conjunto de regras que estabelece de maneira precisa as transformações de um sistema

taca aspectos dos computadores que sugerem um número restrito de situações onde o mundo real poderá ser modelado na máquina.

Não acreditamos que considerações sobre a natureza do computador inviabilize a conceituação dada pelos pesquisadores escandinavos, mas seria útil que eles se posicionassem sobre a questão. Nygaard deu alguns passos nessa direção ao definir *informática* [Ny 86]: “Informática é a ciência que tem como domínio os aspectos de informação dos fenômenos na natureza e na sociedade”.

2.1.2 A conceituação e linguagens

Beta é uma linguagem baseada na conceituação de programação por objetos através de modelos físicos. Fazemos um pequeno estudo da linguagem Beta no capítulo 4.

De uma maneira geral, a conceituação requer linguagens que possam descrever modelos físicos como foram definidos. Devem existir mecanismos para descrever objetos, padrões, classificação, agregação, generalização. Deve ser possível associar seqüência de ações individuais a objetos e deve haver suporte para concorrência. Muitas linguagens ditas orientadas por objeto têm essas características, mas poucas cobrem todas.

Madsen e Knudsen acreditam que mecanismos apropriados para expressão de modelos físicos devem estar presentes em qualquer linguagem de largo espectro, juntamente com mecanismos que forneçam suporte às idéias de programação funcional e programação em lógica.

2.2 Conceituação através de objetos e mensagens

Muito pesquisadores acham que o paradigma da programação orientada por objetos deve ser definido através de dois conceitos básicos: objetos e mensagens. Esse enfoque está intimamente ligado à filosofia de desenvolvimento do sistema *Smalltalk* que buscava obter um sistema a partir de um mínimo de partes intercambiáveis; essas partes deveriam ser tão gerais quanto possível e todas as partes deveriam ser encaixadas em uma estrutura uniforme.

Vejamos três apresentações possíveis do paradigma baseadas em objetos e mensagens:

1. David Robson, da equipe de desenvolvimento do sistema Smalltalk , definiu [Ro 81]:

Ao invés de usar duas entidades para representar a informação (*estrutura de dados*) e sua manipulação (*procedimentos*) independentemente, um sistema orientado por objetos tem um único tipo de entidade, o **objeto**, que representa ambos. Como informação, objetos podem ser manipulados. Como procedimentos, objetos podem descrever manipulações. Informação é manipulada enviando-se uma mensagem para o objeto representado a informação.

Objeto: um pacote de informação e descrição de sua manipulação
Mensagem: especificação de uma das manipulações do objeto
Classe: a descrição de um ou mais objetos similares
Instância: um objeto descrito por uma classe em particular

Todo o processamento é baseado em objetos que trocam mensagens. É feita uma diferenciação entre um objeto e sua descrição (instância e classe). David introduz o conceito de *herança* como uma das maneiras de se compartilhar informação e de se utilizar classes já definidas para criação de novas. Embora *herança* seja uma importante ferramenta de desenvolvimento, não é a questão central nessa exposição do paradigma.

2. Stefik e Bobrow fornecem em [SB 84] a seguinte definição:

O termo “programação orientada por objetos” tem sido utilizado com diferentes significados que têm em comum o conceito de *objeto*. Objetos são entidades que combinam as propriedades de procedimentos e dados, uma vez que fazem computação e guardam um estado local. O uso uniforme de objetos contrastam com o uso separado de procedimentos e dados em programação convencional. Toda a ação é realizada através da troca de mensagens entre objetos.

Herança oferece um meio fácil para a criação de objetos que são *quase* iguais a outros objetos existentes a menos de pequenas mudanças incrementais. Herança reduz a necessidade de especificar informação redundante e simplifica atualizações e modificações, uma vez que a informação pode ser introduzida em um único lugar.

Stefik e Bobrow ligam os conceitos de objetos e mensagens com os conceitos de abstração de dados, polimorfismo, modularidade e herança². Mas, novamente, esses conceitos não são a base do paradigma; são levantados como “lucros” da utilização da visão uniforme de objetos.

²Esses conceitos são abordados no capítulo 3

3. No livro de Tadao Takahashi [Ta 88] encontramos uma conceituação do paradigma baseada em um *modelo de computação* e uma *filosofia de desenvolvimento*. O modelo de computação é o mundo povoado por objetos (e apenas objetos) que trabalham e cooperam entre si, através da troca de mensagens, para realizar uma dada tarefa.

- Objeto

Uma entidade que possui estado interno e um comportamento. O estado interno corresponde a uma memória local que é inacessível e indevassável para outros objetos; o comportamento é definido por um repertório de ações de que o objeto dispõe para responder a demandas (mensagens) externas ou mesmo internas ao próprio objeto.

- Mensagem

Todo processamento é ativado por mensagens entre objetos. Um determinado objeto, recebendo uma mensagem, executará uma ação específica como resposta, alterando seu estado interno, enviando mensagens a outros objetos, criando novos objetos, etc.

- Filosofia de Desenvolvimento

Classes: agrupam objetos com facetas semelhantes. O desenvolvimento orientado a objetos é centrado na identificação de objetos e na construção de uma hierarquia de classes que fatora as propriedades comuns de subclasses e de objetos.

Takahashi diz que a descrição do mundo dos objetos é intencionalmente vaga, devendo ser detalhada para casos específicos.

A maior parte das conceituações existentes para programação por objeto em termos de *objetos* e *mensagens* segue a linha de Robson e Stefik & Bobrow. Certamente essa visão enfoca bem um aspecto central do paradigma — a criação de uma entidade que engloba os conceitos de informação e ação — enfatizando o aspecto de uniformidade do paradigma (objetos que se comunicam através de mensagens). Acharmos muito conveniente a abordagem feita por Tadao ao incluir herança como uma parte central do paradigma, já que organiza o mundo dos objetos. A sua definição para *estado* também parece mais forte ao tornar explícito o aspecto de inacessibilidade do estado interno de um objeto.

As três apresentações não apresentam o conceito de mensagem com detalhe. Ele foi associado com a comunicação entre objetos e a manipulação de informação. Rentsch, em [Ren 82], destaca a necessidade de diferenciar uma chamada de procedimento convencional, que denota uma ação, de um envio de mensagem, que faz uma requisição ao objeto. Através da chamada de um procedimento pode ser exercido algum controle sobre a execução do mesmo (por exemplo, através da passagem de parâmetros). Com o envio de mensagens, a interpretação do desejo do emissor fica totalmente a cargo do objeto que recebe a mensagem; parâmetros podem ser passados (como o nome de

outros objetos a serem envolvidos no processo), mas o objeto receptor mantém sua independência de como atender à requisição recebida.

Entender o mundo dos objetos é mais do que entender as definições de *objeto* e *mensagens*. Uma questão central na programação por objetos é a mudança de visão geral das estruturas; passamos a enxergar a informação pelo lado de fora, preocupando-nos em caracterizá-la ao invés de representá-la. Quando enxergamos a informação “por dentro” pensamos nela através de sua representação em estrutura de dados; sua manipulação é vista sob o ponto de vista de alteração e uso de valores da representação. Ao ver a informação pelo lado de fora passamos a encará-la através de seu comportamento; a manipulação é tida como a junção de ações já previstas pelo comportamento da informação. O conceito de objeto une as formas de caracterização da informação: representação e comportamento.

Pensando na caracterização de informação como parte importante da filosofia do paradigma o conceito de *classe* surge naturalmente. Uma classe contém a descrição de uma porção de objetos parecidos o suficiente para atenderem a uma caracterização comum. Classes, como tudo, são objetos; alguns desses objetos por sua vez podem apresentar características comuns e serem, portanto, agrupadas numa *superclasse*. A idéia de comunalidade entre objetos é, portanto, importante para a criação de um mundo de objetos. A fim de organizar esse mundo procuramos propriedades compartilhadas por vários objetos. O ideal seria conseguir uma fatoração dessas propriedades, de modo que a caracterização de uma propriedade aparecesse em um único lugar. Foi chamado de *herança* o mecanismo através do qual um objeto compartilha com outros uma dada propriedade. O poder de expressão de *herança* aumenta se permitirmos que um objeto reinterprete uma propriedade compartilhada de acordo com suas necessidades. Como estamos em um universo formado exclusivamente por objetos uma propriedade compartilhada é, na verdade, um objeto (ou classe) caracterizado de maneira conveniente.

2.2.1 Prós e Contras

São apontados dois principais pontos favoráveis da concepção de computação dada por objetos, mensagens e compartilhamento de propriedades:

- *uniformidade*

Todos os itens do processamento são objetos. A nenhum objeto é dado um status particular; elementos geralmente tidos como primitivos (inteiros, por exemplo) são objetos como quaisquer outros. Todos os objetos se comunicam usando a

mesma metáfora — troca de mensagens — e referências a objetos são sempre feitas da mesma maneira. Como objetos só podem ser referenciados para fins de trocas de mensagens, nenhum objeto pode ser “aberto”, pois não há como referenciar uma parte do objeto.

Argumenta-se que a utilização de poucas idéias básicas — que podem ser agrupadas em idéias cada vez mais complexas — é um bom princípio de desenvolvimento de software

- *facilidade de mudanças*

Com o uso do compartilhamento de propriedades através da herança a tarefa de executar pequenas mudanças fica facilitada. Essa talvez não seja uma idéia natural, já que alterar alguma propriedade compartilhada por vários objetos parece perigoso e sujeito a conseqüências imprevistas. A vantagem da compartilhação é que a mudança é feita em apenas um local, garantindo que todo o sistema apresente a mesma caracterização para o conceito alterado. Se não tivéssemos a compartilhação, deveriam ser identificados todos os objetos envolvidos (direta ou indiretamente) com o conceito a ser alterado e várias mudanças teriam que ser realizadas.

Nos casos em que as alterações necessárias envolvem boa parte de sistema, a organização dos objetos em termos de compartilhamento pode facilitar a identificação de quais objetos deverão ter suas caracterizações mudadas.

De novo, o aspecto de uniformidade parece favorável, já que independentemente do nível de alteração do sistema as metáforas de comunicação e referência utilizadas continuam as mesmas.

Grande parte dos trabalhos que enfocam programação por objeto através de mensagens e objetos utiliza conceitos e exemplos do ambiente Smalltalk. Várias vantagens do paradigma são levantadas nesses trabalhos com uma argumentação baseada no modelo Smalltalk para objetos, mensagens e herança. Da mesma maneira, as considerações em termos de eficiência — geralmente desfavoráveis ao mundo dos objetos — estão ligadas a uma particular implementação.

Para fazer uma análise crítica da apresentação do paradigma sem entrar em questões de implementação vamos nos ater a examinar duas questões:

1. *A eficácia em definir um estilo de programação*

Para que o mundo formado por objetos que se comunicam através de mensagens venha a refletir um estilo de programação é necessária uma caracterização para a entidade objeto e a inclusão da idéia de herança (compartilhamento) como ponto central do paradigma.

Além de apresentar um objeto como a componente básica da programação que une *dados* e *procedimentos* deve-se deixar evidente a existência de um *estado local ao objeto* que pode ser manipulado exclusivamente pelos seus procedimentos

através de mensagens. Herança deve ser apresentada como a maneira de estruturar o mundo dos objetos baseada na relação existente entre as informações a serem tratadas pelo software. Há bons trabalhos que não seguem essa linha, optando por uma apresentação que, ao nosso ver, supõe uma compreensão a priori. Como dizem Goldberg e Robson em [Sm 80]:

“ Cada uma das palavras objeto, mensagem, classe e método é definida em termo da outra; é quase como se o leitor tivesse que saber tudo antes de saber qualquer coisa”

2. Conveniência de um mundo uniforme

Os trabalhos baseados na idéia do mundo de objetos apontam a uniformidade obtida através dessa conceituação como uma vantagem de programação por objetos, já que o leitor pode aprender rapidamente os poucos conceitos presentes e passar a explorar as várias maneiras nas quais esses conceitos são aplicados em um sistema (em geral os trabalhos discutem como os conceitos estão presentes no Smalltalk).

Nos perguntamos se uma diferenciação entre componentes do modelo não seria útil para representar diferenças de estruturas presentes no domínio de aplicação. Nossa experiência inicial mostra que aparecem pelo menos dois tipos de objetos durante a programação de um sistema: os que são derivados diretamente do domínio da aplicação que está sendo desenvolvida e os que aparecem a partir de opções de implementação. Esta diferenciação poderia ser útil para atividades posteriores de manutenção e extensão do software.

Mac Lane, em [La 81], analisa a questão da uniformidade no ambiente da formalização matemática. Segundo ele, uma visão da matemática uniformizada através da *Grand Set Theoretic Foundation*³ atrapalha a compreensão das estruturas e a distinção do papel exercido por cada conceito. Acreditamos que a mesma crítica se aplicaria a uma visão da computação restrita a objetos de mesmo status, não fosse o conceito de herança. Ao estruturarmos o mundo dos objetos a partir de comunalidades de informação e comportamento rompemos com o mundo totalmente uniforme dos objetos, permitindo que se perceba quais são os conceitos (retratados através de objetos) importantes no software.

2.2.2 A conceituação e linguagens

Há duas linguagens que espelham a apresentação feita: Smalltalk e Actors, abordadas no capítulo 4. Uma linguagem construída para trabalhar em um mundo de objetos deve:

³Matemática é o que pode ser desenvolvido através de regras lógicas de demonstração a partir dos axiomas de Zermelo-Fraenkel para teoria dos conjuntos

- permitir a descrição de uma classe em termos de informação e interface para manipulação. A linguagem deve ser projetada de maneira que a informação só possa ser manipulada pelas rotinas do próprio objeto. A própria descrição deve ser encarada como um objeto do sistema;
- prover um mecanismo de envio de mensagens;
- permitir o compartilhamento de informações entre as classes;
- manter a uniformidade de tratamento entre objetos quaisquer do sistema;

A grande dificuldade é conseguir um sistema onde o implementador e o usuário disponham da mesma visão.

2.3 Conceituação baseada em reutilização de software

Há trabalhos que introduzem o paradigma de orientação por objetos sem criar um novo modelo de computação. A construção do paradigma é feita através de idéias já presentes no ambiente de desenvolvimento de software, buscando disciplina na programação de software que viabilize a reutilização de software já pronto.

Vários trabalhos apresentam o paradigma motivados pelo que Kerr chama de *analogia de engenharia de software* [Ke 86]. Eles constroem uma caracterização para programação por objetos ao atacar a seguinte questão [Mey87a]:

“Por que software não é como hardware? Por que o desenvolvimento em software sempre começa do zero? Deveriam haver catálogos de módulos de software como há catálogos de componentes VLSI: quando construímos um novo sistema deveríamos apenas encontrar as componentes convenientes nesse catálogo e combiná-las, ao invés de toda vez reinventar a roda. Escreveríamos menos software e provavelmente faríamos melhor a parte que realmente desenvolvêssemos. Então os problemas que nós todos lamentamos — alto custo, prazos, falta de confiabilidade — desapareceriam. Por que não é assim?”

Um paralelo entre a engenharia e o desenvolvimento de software é usualmente traçado em torno das atividades de planejamento, projeto, desenvolvimento, análise de custos, controle do produto e administração do projeto. Kerr estende essa análise em

[Ke 86], incluindo o que ele considera um aspecto fundamental da analogia software/engenharia mecânica⁴: o *material*. O objetivo da análise é encontrar qual a influência do uso de materiais no fato de que sistemas mecânicos são mais confiáveis do que sistemas de software.

Materiais possuem características físicas que permitem obter uma impressão intuitiva de seus custos, complexidade e verificação de estado. Eles têm características intrínsecas de resistência, força e flexibilidade. Em engenharia mecânica, a *forma* é fundamental, pois componentes que serão utilizadas juntas precisam de uma especificação de interface comum, ou seja, de uma forma apropriada para o encaixe. A maneira como são conectadas componentes também é fundamental: ligações longas e tortuosas tendem a ser caras, ineficientes e não confiáveis. Essas propriedades permitem que tiremos algumas conclusões:

- Imposições físicas guiam a maneira de conectar componentes em engenharia mecânica, filtrando configurações absurdas ou excessivamente complexas. No caso do projeto de software a distribuição topológica das componentes é muito menos restrita;
- Nenhum sistema mecânico é composto predominantemente de componentes críticas frágeis. Já no software o mais insignificante problema em qualquer componente usualmente tem consequências desastrosas;
- Devido à possibilidade de caracterizar em engenharia mecânica uma componente através de suas características físicas (além da funcionalidade) torna-se muito mais razoável utilizar padrões já existentes para uma componente do que inventar uma nova. Além de baixar o custo do desenvolvimento do novo sistema, a chance de que este apresente falhas decorrentes da componente padrão utilizada (geralmente já bem testada) diminui bastante. Na produção de software, no entanto, grande parte do esforço é empreendido no desenvolvimento de rotinas já feitas e utilizadas várias vezes antes (muitas vezes pela mesma equipe de desenvolvimento).

A partir dessas considerações *programação orientada por objetos segue como uma maneira de tratar componentes em software*. O enfoque é fortemente baseado na reutilização de software como solução para os problemas existentes. O apêndice A focaliza os conceitos de reutilização de uma maneira mais geral do que o reaproveitamento de componentes. Uma justificativa de programação por objetos dada pela analogia hardware/software ou mecânica/software não é exclusividade dessa terceira conceituação a ser apresentada. Adeptos do mundo de objetos e mensagens também introduziram o conceito de software-ICs (*software integrated circuit*, uma componente reutilizável de software) em [C-L 85].

⁴o termo engenharia mecânica é usada para abranger as áreas da engenharia que tratam da construção de sistemas não abstratos

2.3.1 Apresentação do paradigma

Programação por objetos é uma técnica de decomposição do software:

Decida que classes você quer, especifique um conjunto de operações para cada classe, faça com que aspectos comuns entre classes sejam evidenciados.

As componentes do software são as classes, que correspondem a entidades do domínio da aplicação. O conceito de classe engloba o comportamento da entidade associada através da interface. A execução de um programa se baseia na criação, eliminação e manipulação de objetos (instâncias de uma classe) exclusivamente através das operações definidas para a classe.

Segundo Kerr [Ke 86], a distinção entre sistemas tradicionais e orientados por objetos é que, no último, adotamos uma visão muito menos fragmentada de estruturas de dados e seqüências de ações sobre dados; estes dois aspectos da informação passam a ser encarados como intimamente relacionados.

O trabalho de Stroustrup [?] define programação por objetos com esse enfoque, apresentando-a como uma evolução do paradigma de programação por procedimentos em direção à obtenção de componentes melhor definidas e que facilitem as tarefas de programação e depuração.

As várias componentes do software (classes) podem apresentar estrutura e comportamento semelhantes. É feita a fatoração de maneira a captar as partes em comum e sempre tendo em vista o objetivo de obter componentes bem definidas e que façam sentido numa visão geral de encaixamento dessas partes. O mecanismo para relacionar as classes (geralmente em termos de especialização da componente) é chamado herança.

Exemplifiquemos essa visão de programação por objeto: na Teoria dos Grafos fortes candidatos a componentes são as classes *grafos*, passeios, circuitos, conjuntos, etc. Assim, o grafo de Petersen é um objeto da classe *grafo* e o conjunto de seus vértices é um objeto da classe *conjunto*. Para implementar uma classe se deve escolher uma estrutura de dados conveniente e definir quais as operações que podem ser realizadas em um objeto da classe. Observe que as classes de passeios e circuitos apresentam alguma comunalidade (ambas são seqüências de vértices e arestas). A aplicação do paradigma deve explicitar essa relação entre as classes através de uma classificação (herança).

2.3.2 Prós e Contras

O aspecto positivo dessa forma de apresentação é explicitar os objetivos do paradigma: obtenção de software a partir de componentes bem definidas. Através desse enfoque programação por objetos justifica-se em termos de reutilização de software com modularização conveniente e extensão.

Consideramos que a forma de apresentação em cima de um estilo e idéias um tanto vagas como interface e comunalidade de classes não resolve bem o problema de caracterização de programação por objeto, pois não elimina a possibilidade de interpretações baseadas em impressões pessoais.

2.3.3 Linguagens

Para essa linha de pesquisadores a linguagem utilizada não é o mais importante, já que as idéias poderiam ser utilizadas em quaisquer linguagens, ficando a idéia de componente sob a responsabilidade da disciplina do programador. O uso de linguagens que impedem um desvio da idéia básica de componente como uma *caixa preta* (por exemplo, impedindo que funções outras que as definidas na interface da classe acessem a estrutura de dados) facilitam a fase de depuração do sistema e aumentaria sua confiabilidade. Exemplos de linguagens baseadas no princípio de reutilização de software e componentes são C++ e *Eiffel*, abordadas no capítulo 4.

2.4 Discussão sobre as três apresentações

Embora nós tenhamos classificado as apresentações baseados em suas origens (derivados da linguagem Simula ou não), a mesma divisão reflete uma diferenciação mais forte: as duas primeiras apresentações sugerem uma nova maneira de pensar na execução de um programa enquanto que a última prefere destacar uma disciplina de programação.

As diferentes conceituações envolvem, em última análise, a mesma idéia: a criação de uma unidade que envolve dados e comportamento desses dados. Nos modelos físicos (seção 2.1) a unidade recebeu o nome de *objeto*, os dados foram chamados *atributos* e o comportamento era retratado através das ações. No mundo dos objetos e mensagens (seção 2.2) novamente a unidade foi chamada de objeto; os nomes para dados e comportamento são fornecidos em função do Smalltalk (*variáveis de instância* e *métodos*). Na visão motivada pela reutilização de software (e baseada, no fundo, em conceitos de tipos abstratos de dados) usa-se *classe* para denominar a entidade que descreve estruturas e comportamento de objetos.

Achamos que as três apresentações cumprem seu papel de introduzir as idéias

básicas de programação por objeto, mas causam diferentes impressões iniciais. A “revolução” do conceito de computação dadas pela primeira e segunda apresentação podem levar as pessoas envolvidas em grandes sistemas (que já vem sendo desenvolvidos há algum tempo) a considerar inviável um tratamento novo na produção de software por julgá-lo incompatível com o que vinha sendo desenvolvido; a terceira apresentação, ao pôr as coisas em termos de disciplina, incentivaria a adoção do novo paradigma (sob o risco, talvez, de que a disciplina desejada não fosse atingida em função de vícios adquiridos com a forma anterior de desenvolver software). Por outro lado, pessoas empenhadas no desenvolvimento de sistemas novos e não convencionais acharão a terceira apresentação menos atraente do que as outras.

Nós vemos outra diferença entre os três enfoques com relação à metodologia de desenvolvimento sugerida, à primeira vista, por cada apresentação. Os modelos físicos guiam o desenvolvimento, pois a apresentação foi feita em termos de como abstrair um modelo. O mundo dos objetos tem uma apresentação mais vaga da metodologia, pois não faz a ligação entre objetos e domínio de aplicação (há trabalhos que dizem apenas que os objetos devem espelhar os objetos do mundo real que temos em mente); quando esta apresentação é feita usando o Smalltalk há a impressão de que é necessário conhecer muita coisa da biblioteca Smalltalk antes de se desenvolver qualquer coisa. A terceira apresentação sugere como metodologia que se identifique as componentes básicas do sistema, tornando-as unidades; passa a impressão de que metodologias usualmente utilizadas (Métodos de Jackson, Warnier, etc) continuarão a nortear o desenvolvimento, atadas agora à idéia de obtenção de *caixas pretas* e mais o trabalho de gerenciar a coleção de *caixas*.

A existência de diferentes maneiras de introduzir programação por objeto certamente desfavorece um aperfeiçoamento mais rápido do paradigma, já que há referências a um único conceito através de termos distintos e falta de consenso sobre o significado de algumas palavras-chave da área.

Na nossa experiência com programação por objeto uma visão unificadora dos três enfoques só surgiu com a aplicação dessas idéias no desenvolvimento de um sistema.

Capítulo 3

Conceitos e Formalização

Neste capítulo estudamos separadamente cada um dos conceitos usualmente associados à programação por objetos. Algumas das idéias apresentadas têm sido estudadas há muito tempo fora do contexto do paradigma dos objetos; para estas nos preocuparemos apenas em fornecer uma descrição conveniente à nossa abordagem. Já os conceitos intimamente relacionados com o novo paradigma (como *classe*, *herança* e *polimorfismo*) serão estudados em detalhe.

3.1 Idéias básicas em programação por objetos

Abordaremos os pontos fundamentais ao paradigma tentando identificar as origens dos conceitos (em termos de motivação) e a capacidade de cada um deles em retratar peças chaves no desenvolvimento e execução de programas.

3.1.1 Tipos

Devido à influência das linguagens de programação tradicionais (Fortran, Algol, C, Pascal, etc) o termo *tipo* foi relacionado com as alternativas de armazenamento (e respectivas operações) que vêm embutidas em uma linguagem. Por exemplo, em Pascal *integer* e *boolean* são tipos da linguagem que podem ser manipulados através de operadores como *+* e *and*, respectivamente. Em Pascal, há a possibilidade de definição de um novo tipo (*type*), mas a definição do usuário não terá o mesmo status que os tipos básicos; por exemplo, apenas os tipos pré-definidos da linguagem podem ser utilizados em expressões envolvendo operadores. Adotar a noção de tipos sugerida pelas construções de uma linguagem não fornece um significado geral para *tipos* em linguagens de programação, já que os tratamentos que as diversas linguagens fazem da relação *dado-tipo* diferem bastante.

O que são tipos?

O trabalho de Cardelli e Wegner [CW 85] investiga a origem da necessidade de tipos em linguagens de programação. Eles citam alguns universos que se organizaram através dos tipos: o universo das cadeias de bits, das S-expressões do Lisp puro, das expressões- λ do λ -cálculo, dos conjuntos da teoria dos conjuntos. No caso do universo das cadeias de bits na memória do computador, por exemplo, observamos que todos os elementos do universo são do mesmo tipo: uma cadeia de bits com tamanho fixo (*palavra*). Ao olhar para uma sequência de palavras da memória não poderemos, em geral, dizer o que está sendo representado. O significado da cadeia de bits é determinado por uma *interpretação externa* de seu conteúdo.

Ao trabalhar em qualquer universo sem tipos começamos a organizá-lo de maneira conveniente ao nosso propósito. Tipos surgem informalmente em qualquer domínio a fim de classificar os elementos de acordo com o uso e comportamento. No caso do universo de cadeias de bits podemos querer distinguir caracteres de operadores, embora ambos sejam em última análise cadeias de bits. O universo dos valores que aparecem em uma computação (obtidos através de interpretações para cadeias de bits) se decompõe naturalmente em subconjuntos formados por elementos de comportamento uniforme. É comum referir-se a cada um desses subconjuntos como sendo um tipo. O universo dos bits foi organizado em conjuntos de valores e operações. Passamos a enxergar este universo como sendo um universo com tipos (inteiros, reais, caracteres, etc) mas, num certo sentido, paramos no meio do caminho entre universos sem e com tipo, já que é muito fácil violar os tipos que criamos apelando para o universo subjacente das cadeias de bits.

O principal objetivo da organização de um universo através de tipos é impor restrições sobre a interação de elementos. No caso do universo das cadeias de bits (manipulados através de programação) os tipos contribuem para a corretude ao proteger os objetos de inconsistências devidas a manipulações e interações entre objetos.

Uma conceituação de tipos conveniente para programação por objetos se baseia no fato de que os valores que utilizamos em uma computação são vistos levando em consideração seu comportamento, mas utilizamos um universo que não possui categorias. Tipos podem então ser vistos como um conjunto de vestimentas —funcionando como uma armadura — que protege a representação de usos arbitrários ou não intencionais. Eles provêem uma cobertura para a representação que restringe a maneira como os objetos interagem e relaciona o comportamento do elemento da computação e sua representação. Violar um sistema de tipos significa operar diretamente sobre a representação.

Esta apresentação para tipos pode parecer muito restritiva e distante da nossa experiência com linguagens de programação tradicionais. Acreditamos que isto se deve principalmente à associação que há muito tempo se fez entre *eficiência de um programa e aproveitamento máximo da forma de representação*; com essa filosofia,

tipo era visto apenas como uma especificação da forma de armazenamento para valores ou uma maneira de criar imposições sintáticas sobre uso de operadores.

Adotaremos para tipo a noção de um conjunto de valores com o mesmo comportamento. A descrição de um tipo em uma linguagem de programação seria, então, uma maneira de relacionar representação e comportamento evitando que valores que apareçam na computação sejam manipulados de forma inconsistente com o seu comportamento.

3.1.2 Tipos Abstratos de dados

Uma abstração é uma maneira de representar, em um único elemento, um grupo de coisas de maneira a expressar suas similaridades e suprimir suas diferenças [Ho 83]. Durante muito tempo o trabalho de abstração na atividade de desenvolvimento de programas restringiu-se a analisar como alguma ação é feita (procedimento). Programar usando *tipos abstratos de dados* significa usar abstração para classificar as informações manipuladas no processamento com base no comportamento destas.

O estilo de programação com tipos abstratos de dados propõe que os valores (dados) a serem usados na computação sejam analisados e agrupados (a partir de semelhanças de comportamento) de maneira a obter uma descrição abstrata que independa da forma escolhida para representação desses valores. Assim, antes de começar a atividade de programação das ações que atenderão aos requisitos do sistema devemos analisar as informações manipuladas, abstrair propriedades que permitam classificá-las e descrever o comportamento de cada classe através de uma lista completa de manipulações permitidas. Só então seriam escolhidas as estruturas de armazenamento e codificadas as rotinas que implementam o sistema.

A utilização de *tipos abstratos de dados* representa uma forte guinada em direção a um universo com tipos; suas idéias eliminariam da programação a possibilidade de encarar valores como seqüências de bits, pois cada valor da computação foi associado a um comportamento e portanto só poderia ser manipulado através das operações previamente previstas pelo comportamento.

Dijkstra, em artigo publicado em 1979 [Dijk], tece considerações sobre eventuais limitações dos tipos abstratos de dados: “não é precisamente um exagero afirmar que, após cinco anos de intensa pesquisa e desenvolvimento, *pilha* ainda é o único tipo abstrato de dados já projetado”. Nós acreditamos que agora, com o uso de programação por objetos, muitos tipos abstratos de dados foram efetivamente projetados. Através da utilização do paradigma dos objetos podemos notar que *tipos abstratos de dados* são utilizáveis na prática de uma maneira muito natural. Mas essa experiência evidencia também o surgimento de alguns problemas quando tentamos dividir as informações em categorias, pois as classes obtidas não são necessariamente disjuntas.

3.1.3 Encapsulamento

Uma efetiva utilização de tipos abstratos de dados pressupõe uma certa inviolabilidade da representação escolhida para os valores de um tipo. Assim, a utilização dos tipos abstratos de dados deve seguir um princípio básico: *encapsulamento*. Encontramos em trabalhos referentes a tipos abstratos de dados maneiras um pouco distintas para definição de encapsulamento. Veloso [Ve 86] define encapsulamento como “representação dos objetos e implementação das operações abstratas reunidas em uma única região do programa, de preferência em uma só unidade sintática”. Para expressar a idéia básica de *tipos abstratos de dados* Veloso usa, além de encapsulamento, o conceito de proteção: “os objetos só podem ser manipulados por meio das operações de um elenco pré-estabelecido, fazendo com que a representação fique invisível e protegida contra manipulações indisciplinadas”.

Nós adotaremos para encapsulamento uma conceituação que já engloba proteção. A intenção é denotar, através de encapsulamento, uma parede que protege a representação ao conceder acesso apenas às operações definidas para o tipo abstrato de dados.

Snyder, ao definir encapsulamento em [Sny 86], destaca o importante papel do encapsulamento no desenvolvimento de software:

Encapsulamento é uma técnica para minimizar inter-dependência entre módulos escritos separadamente através da definição de uma interface externa estrita. A interface externa de um módulo serve como um contrato entre o módulo e seus usuários e portanto entre os implementadores desse módulo e o resto da equipe de desenvolvimento. Se os usuários dependem apenas da interface externa o módulo pode ter sua implementação alterada sem afetar os demais usuários, desde que a nova implementação suporte a mesma interface (ou uma compatível). Assim os efeitos de mudanças compatíveis podem ser confinados.

3.1.4 Classes e objetos

Como já vimos, o uso de *tipos abstratos de dados* permite classificar os valores da computação em categorias de mesmo comportamento. Resta esclarecer qual é o papel dessas abstrações no software, ou seja, o que fazemos com cada um desses *tipos* durante a programação.

Um tipo, sendo um conjunto de valores aliado a um comportamento, não é propriamente uma entidade abstrata. Queremos embutir em nossos programas uma descrição do tipo, ou seja, o *tipo abstrato de dados* que define o comportamento do conjunto de valores através da especificação de uma lista de operadores. Em *programação por*

objetos um tipo abstrato de dados é usualmente chamado de *classe*. Assim, *classe* denota a descrição de um *tipo* através da enumeração das manipulações permitidas.

Como veremos no capítulo 4, as linguagens de programação assumem uma conotação mais concreta para classe, pois a descrição do tipo passa a englobar também a especificação da forma de armazenamento escolhida. Dessa forma, grande parte da responsabilidade pela independência da abstração em relação à representação recai sobre o programador. Isto não significa que o encapsulamento esteja comprometido, mas apenas que a descrição do comportamento pode estar excessivamente atrelada à escolha da estrutura de dados.

Durante o nosso trabalho adotaremos para classe a idéia de uma unidade de software responsável pela manipulação de certas espécies de valores. Pelo princípio do encapsulamento estas unidades são indevassáveis; as enxergaremos como *caixas pretas*. O funcionamento interno dessas unidades é conhecido apenas pela equipe que construiu e mantém a caixa, ou seja, seus implementadores. As operações disponíveis para a classe constituem a interface da caixa, que poderá ser utilizada pelas demais unidades do software — seus *usuários* — exclusivamente através das manipulações previstas pelos seus implementadores. Os usuários da caixa preta, não dispendo de informação alguma sobre aspectos internos de projeto da caixa, estão impossibilitados de tirarem proveito de formas particulares de armazenamento dos valores manipulados.

Um valor criado e manipulado por uma classe durante a computação é chamado de *objeto*. Um objeto, sendo um valor, é um elemento do conjunto especificado pela classe e portanto está associado a um tipo.

Uma outra possibilidade para definirmos *objeto* é encará-lo como sendo uma instância (real) da classe (abstrata) que o define. Dessa maneira a noção de objeto fica mais próxima do conceito de uma variável em linguagem de programação, já que especifica uma área de armazenamento cujo "formato" e forma de manipulação estão definidos pela sua classe; a classe funciona como um padrão a ser seguido para criação e manipulação dos valores de um dado tipo. Observe que esse enfoque desvincula um pouco o *objeto* do *elemento do conjunto de valores* que ele representa, pois vários objetos poderiam ser criados para armazenamento e manipulação do mesmo valor.

Acreditamos que sob o ponto de vista de formalização do paradigma a conceituação adequada para o vocábulo objeto é a que denota um valor associado a uma classe e

portanto a um tipo: simplesmente um elemento (que aparece durante a execução de um programa) de um conjunto já especificado. O aspecto semântico da computação independe da forma de armazenamento, pois o interesse maior é por uma descrição precisa do *efeito* de cada operador¹ da classe sobre os valores utilizados para representar o domínio da aplicação.

Quando nos voltarmos para linguagens de programação e metodologias veremos que a conceituação adotada para objeto é uma questão delicada que influencia a concepção e o projeto de software.

Alguns exemplos com tipos, classes e objetos

A visão conceitual de *tipos* como conjunto de valores com operações associadas, *classes* como descrição desses tipos e *objetos* como instâncias dos tipos criadas a partir dessas descrições fornece uma boa base para formalização do paradigma e permite que seja analisada a capacidade de uma linguagem de programação em lidar com esses conceitos. Façamos um parêntese na abordagem “teórica” a fim de manter o pé no chão: esses conceitos não eliminam problemas do desenvolvimento de software como, por exemplo, o aparecimento de conjuntos de valores com difícil caracterização. Focalizaremos esse problema no capítulo 5 ao discutirmos sobre o *uso* do paradigma de objetos, mas antes vejamos alguns exemplos:

- *Racionais*

O conjunto Q dos números racionais é um candidato natural para muitos tipos diferentes, basta que lhe associemos as operações desejadas para o tipo. Os elementos do conjunto apresentam um padrão uniforme de comportamento em relação às operações usuais, podendo portanto ser facilmente caracterizado como um tipo abstrato de dados.

A classe *Rac* será a unidade do software responsável pela manipulação dos racionais utilizados; através dela definimos todas as manipulações desejadas para estes valores:

- as quatro operações aritméticas (o manipulador recebe dois objetos da classe *Rac* e devolve como resultado um objeto da mesma classe);
- operações de comparação entre racionais ($=$, $\neq$, $\geq$, $>$, $\leq$, $<$);
- operações de entrada e saída de racionais.

Para armazenarmos os valores de Q usaremos objetos da classe *Rac*. O conceito de objeto fornece uma visão mais concreta para o valor, instanciando a abstração em algo passível de armazenamento. Esta visão mais próxima da máquina se

¹O conceito de operador será formalizado na seção 3.2 de maneira a denotar uma função cujo domínio envolve o tipo associado à classe

reflete na definição das operações para a classe ao induzir operações envolvendo inicialização e atribuição de objetos.

- *Listas*

Uma abstração do aspecto funcional de uma lista nos leva a um conjunto de operações necessárias para manipulação de uma coleção (por exemplo inserção, remoção e busca de elementos). A definição de uma classe para listas é um exemplo canônico de *caixa preta*, pois a interface pode estar completamente definida sem que pensemos na forma de implementação. Mas o implementador da caixa preta, após escolher sua forma predileta para o armazenamento dos elementos da lista, pode considerar o “*ambiente interno*” da caixa insuficiente para implementá-la, pois durante a codificação das operações sente falta de informações extras sobre o tipo de elemento que será armazenado na lista.

Como veremos com a seção 3.7 e o capítulo 5, a abstração dos valores desejados para um tipo pode não ser suficiente para uma completa descrição e implementação da classe em uma linguagem de programação. Durante a fase de projeto (após a fase de análise do domínio do software) muitos tipos podem ter sua definição alterada ou tipos novos podem surgir.

- *Alunos*

Abstrair a classe que descreve o conjunto de alunos da USP requer uma análise da aplicação que permita um levantamento das características do comportamento dos alunos que são de interesse no software. Temos então um problema de modelagem tradicional, onde precisamos descrever formalmente elementos do mundo real; a abordagem do desenvolvimento de software através da utilização de objetos não torna menos árdua essa tarefa.

Nem sempre é fácil decidir que informação do elemento colocamos na descrição da classe ou em que classes existentes no software encaixamos novas informações. Essas decisões devem ser tomadas cuidadosamente já que refletem a estrutura do software (pois estamos utilizando as classes como unidades de desenvolvimento) e portanto serão cruciais para a manutenção e extensão do sistema.

3.1.5 Herança

As várias classes abstraídas podem apresentar aspectos semelhantes refletindo ligações e dependências entre os dados no domínio da aplicação. Não podemos, portanto, considerar que as caixas pretas que compõem o software são estanques. A grande novidade do paradigma de programação por objetos² foi acrescentar às idéias de tipos abstratos de dados um mecanismo que permite expressar, através das classes, relações entre tipos.

²Há autores que não consideram o conceito de herança indispensável para a utilização do paradigma dos objetos ([Cox 86], [Lad 88], [J-K 87])

Os tipos associados às classes, sendo conjunto de valores, podem possuir elementos em comum e dessa forma indicar que as caixas pretas que implementam esses tipos não são totalmente independentes. Vejamos a relação entre classes em duas situações usuais, a fim de exemplificar a necessidade de relacioná-las:

- Tipos diferentes apresentam alguns valores comuns

Sejam A e B tipos com um valor x em comum e C_A e C_B as classes que implementam estes tipos. Embora x obedeça simultaneamente aos comportamentos descritos por C_A e C_B , não existe necessariamente relação entre os dois comportamentos de uma maneira geral; sequer podemos concluir que uma parte das caracterizações dadas por A e B coincidem. A abstração leva então à construção de “caixas” para os tipos A e B com interfaces distintas. Um objeto o_x criado para representar o valor x durante a computação deverá responder aos dois comportamentos, ou seja, ser manipulados pelas duas classes. Qual das classes criará o objeto? Como um objeto criado por uma classe poderá ser manipulado por outra? Esse impasse pode ser resolvido se a classe responsável por o_x tiver explícita em sua definição uma relação com outras classes com as quais compartilhe elementos.

- Existe uma hierarquia entre tipos distintos

Sejam C e D tipos tais que $C \subset D$.

Todo valor x de C também é um valor de D , ou seja, os valores que seguem a especificação dada por C também podem ser vistos como valores que seguem o comportamento descrito em D . Nessa situação é muito provável que durante a abstração dos conjuntos de valores já fosse possível detectar uma relação entre os dois tipos e perceber que a descrição de D engloba, de alguma forma, a descrição do tipo C . As classes C_C e C_D devem ser criadas tornando explícita a relação entre os comportamentos, de maneira que qualquer objeto criado pela classe C_C possa ser manipulado pelas operações definidas em C_D .

Analisando os tipos que aparecem em um software podemos detectar algumas relações intrínsecas entre as unidades responsáveis por esses tipos, ou seja, gostaríamos que as classes C_A , C_B , C_C e C_D refletissem a relação existente entre os tipos A , B , C e D . Há uma ótica que unifica a relação entre tipos nos dois casos:

A idéia de comunalidade de comportamento revela uma relação hierárquica entre os tipos muito semelhante a relação de “contido” entre subconjuntos quaisquer. Dados os tipos T_p e T_f dizemos que T_f é **subtipo**³ de T_p se $T_f \subseteq T_p$. Os conjuntos T_f e T_p são tipos e portanto estão atrelados a um comportamento uniforme para seus elementos, ou seja, a um elenco de manipulações válidas para os valores do conjuntos. As entidades que descreverão os tipos — as classes — terão que expressar a relação de comportamento associada a relação de tipos. Por exemplo, sendo C_f e C_p os comportamentos de T_f e T_p temos que os valores de T_f respondem tanto a C_f quanto a C_p , não sendo possível então que alguma característica de comportamento apareça definida em C_p e não em C_f . Assim, o comportamento C_f engloba o comportamento C_p e portanto a classe que implementar o tipo T_f terá que implementar aspectos comportamentais presentes também na classe correspondente a T_p . A figura anterior representa as relações “ C é subtipo de D ”, “ $A \cap B$ é subtipo de A ” e “ $A \cap B$ é subtipo de B ”.

O conceito de herança surge para denotar o relacionamento entre classes. Através desse mecanismo é possível informar que o comportamento descrito por uma classe engloba aspectos já definidos em outra. Ao relacionarmos classes através da herança estaremos explicitando acumulação de comportamento; a unidade que implementa um tipo pode então ser utilizada para definição de novas unidades, não estando mais isolada num amontoado de caixas pretas. Esta idéia introduz uma hierarquia que organiza as unidades do software responsáveis pelos objetos manipulados durante a execução e explicita uma relação de derivação entre as várias classes. A definição de uma classe passa a englobar, além da lista de operadores que define seu comportamento, a lista das classes que serviram de base para a especificação da nova unidade.

Dizemos que uma classe F é subclasse da classe P se F é construída a partir da classe P . Isto significa que o comportamento de um objeto da classe F é constituído pelas operações especificadas para P e pelas novas operações definidas por F .

Através da herança uma classe pode oferecer às suas subclasses tudo aquilo que possui: uma descrição de comportamento através de uma lista de operadores. Se uma classe C_{Filho} herda de uma classe C_{Pai} ela terá o direito de dispor das operações definidas para C_{Pai} , ou seja, poderá utilizá-las ou modificá-las para seu próprio uso. Assim, C_{Filho} compartilha do comportamento de C_{Pai} podendo ainda acrescentar características novas que diferenciam as duas classes.

Denotamos a relação de subclasse através do seguinte grafo:

³Considere esta definição apenas como uma noção intuitiva. A relação de subtipo será formalizada na seção 3.2

F é subclasse de P e portanto herda o seu comportamento. Um objeto da classe P pode ser manipulado pelas operações $p_1, p_2, \dots, p_m$; um objeto da classe F pode ser manipulado pelas operações $f_1, f_2, \dots, f_n$ e $p_1, p_2, \dots, p_m$. Observe que alguns operadores de F podem ser simplesmente uma redefinição conveniente de operadores de P .

O mecanismo de herança se mostra conveniente para relacionar os tipos nos dois casos abordados ($A \cap B \neq \emptyset$ e $C \subseteq D$). A construção de unidades responsáveis pela manipulação dos valores presentes em A , B , C e D poderia ser feita sob a seguinte forma:

- Para A , B :

1. Cria-se uma classe C_A para o tipo A e uma classe C_B para o tipo B , ou seja, são especificadas as rotinas que compõem o comportamento de cada um dos tipos;
2. Abstrai-se características que delineiam o conjunto $A \cap B$. Se o comportamento desse conjunto nada mais é do que a união das manipulações já especificadas para as classes A e B bastará criar uma classe $C_{A \cap B}$ derivada das classes C_A e C_B para que tenhamos unidades responsáveis pelos valores dos conjuntos A e B . Observe que um objeto criado para armazenar um valor comum a A e B será gerado sob responsabilidade da classe $C_{A \cap B}$ e portanto poderá assumir simultaneamente as características herdadas de C_A e C_B . Se, por outro lado, o comportamento do conjunto $A \cap B$ possuir peculiaridades não presentes nos conjuntos A e B isoladamente poderemos, analogamente, construir uma classe $C_{A \cap B}$ que adicionará características de comportamento as já especificadas em C_A e C_B .

- Para C , D :

1. Especifica-se a classe C_D , responsável pelo conjunto de valores D ;

2. Cria-se a classe C_C como subclasse de C_D , assim os aspectos comuns entre C e D já estão definidos bastando então acrescentar características de comportamento exclusivas a C .

O que podemos expressar através de herança

A impressão inicial que uma abordagem conceitual para herança nos passa é de que esta tem como objetivo expressar hierarquias existentes entre tipos. Na verdade, durante o desenvolvimento de software a utilização mais usual para herança é permitir que caixas pretas já construídas sejam usadas para obtenção de novas, conseguindo então reutilização de operações já implementadas. Muitos autores preferem abordar herança exclusivamente sob este ponto de vista; por exemplo, Cox em [Cox 86] define:

Herança é uma ferramenta para organizar, construir e usar classes reutilizáveis. Sem herança, toda classe seria uma unidade isolada, desenvolvida a partir do zero. Classes diferentes não teriam nenhum relacionamento entre si, já que o projetista de cada uma supre a classe com as operações que escolher. Qualquer consistência entre as classes é resultado de disciplina por parte dos programadores.

Consideramos que o problema desse enfoque é que, a menos que sejam impostas restrições aos mecanismos de reaproveitamento de unidades, as classes derivadas com o único intuito de reaproveitar rotinas estarão descrevendo tipos sem relação com o domínio da aplicação, prejudicando assim o nível de abstração do software e conseqüentemente tornando obscura a forma de conexão entre as componentes do sistema.

O trabalho de Halbert e O'Brien [H-O 87] é um excelente estudo sobre as formas de utilizar tipo e herança na produção de software, exemplificando possíveis formas de encaixar a derivação de unidades em uma metodologia de desenvolvimento (abordaremos uma visão mais geral do poder de expressão de herança no capítulo 5).

Um exemplo com classes e herança

A maioria dos trabalhos introduz o conceito de herança através de exemplos, o que tem a vantagem de evidenciar a utilidade da interligação de classes mas acaba prejudicando uma análise do poder de expressão do mecanismo e escondendo, através de construções de linguagens, os problemas que aparecem com a redefinição de operadores. Para ilustrar como a relação de subclasse e herança podem ser utilizadas para ligar unidades do software veremos um exemplo bem simples e nada original, mas que já denuncia alguns pontos importantes no uso de herança no desenvolvimento de programas.

Consideremos os conjuntos R , que engloba todos os retângulos, e Q , que denota todos os quadrados.

Os elementos de R e Q são presenças usuais em programas que manipulam figuras geométricas. Esses conjuntos captam elementos com características bem definidas e com comportamento passível de descrição através de uma lista de manipulações desejadas no software. Assim, é conveniente que tenhamos os conjuntos R e Q como tipos. Para construirmos as unidades do sistema responsáveis pela manipulação de quadrados e retângulos devemos abstrair as características de R e Q que são de interesse ao software e descrevê-las através de classes.

Definimos uma classe *Ret* que descreve em sua interface as manipulações previstas para retângulos:

O implementador de *Ret* deverá encontrar uma forma conveniente para armazenamento de retângulos e codificar cada uma das operações da interface. Uma vez concluída a implementação de *Ret* seus usuários poderão criar objetos que armazenem retângulos e manipular esses objetos usando (apenas) as manipulações permitidas.

Para definirmos a classe *Quad* não é necessário começar o trabalho do zero: sabemos que $Q \subset R$ e portanto que o comportamento a ser descrito em *Quad* engloba o descrito em *Ret*. Elementos do tipo *quadrado* são também elementos do tipo *retângulo* e portanto incluem em sua caracterização toda a descrição feita em *Ret*. Temos então a seguinte relação entre as duas classes:

A classe *Quad* é subclasse de *Ret*, herdando todas as operações definidas para *Ret*. Ao criarmos *Quad* como subclasse de *Ret* estamos automaticamente dotando *Quad*

das operações (*translação, rotação, mudança de escala, cálculo de área, cálculo de perímetro, impressão*) definidas em sua superclasse. Ao manipularmos um objeto da classe *Quad* com a operação *translação*, por exemplo, será utilizada a rotina de *translação* definida pelos implementadores de *Ret*. Isto certamente torna mais fácil a tarefa dos responsáveis por *Quad*, que não precisam se preocupar com *translação, rotação, etc.* Mas o reaproveitamento de operações já definidas pode não ser conveniente algumas vezes. Por exemplo, é possível que por uma questão de eficiência o responsável pelos quadrados prefira suprir a classe *Quad* com sua própria operação de *cálculo do perímetro* (a fim de aproveitar o fato de todos os lados de um quadrado apresentarem o mesmo comprimento), desprezando assim a rotina que herdou de *Ret*. Passaríamos então a ter a seguinte estrutura de classes para manipulação de retângulos e quadrados:

Suponhamos que queiramos aumentar a interface de *Ret* acrescentando mais uma manipulação para retângulos: *Quebra*. Essa operação age sobre um retângulo partindo-o ao meio (obteríamos então como resultado da operação dois retângulos). *Quad*, ao herdar essa operação, passaria também a dispôr de uma operação que age sobre um quadrado partindo-o ao meio e obtendo como resultado dois retângulos.

Herança assume um papel central no aspecto evolutivo do sistema e a hierarquia escolhida para ligar as unidades influenciará de forma determinante o bom andamento do projeto, assim como a eficiência de seu funcionamento e manutenção.

3.2 Apresentação formal dos conceitos

Muitos pesquisadores têm se dedicado ao estudo formal do paradigma buscando desenvolver teorias de tipos para objetos e suas interações. Acredita-se que uma conceitualização sólida favorecerá a derivação de sistemas, possibilitará verificação de funcionamento e contribuirá para a obtenção de implementações eficientes.

O trabalho de Danforth e Tomlinson [D-T 88] apresenta de maneira excelente várias abordagens para tipos abstratos de dados e herança, obtendo assim possíveis embasamentos formais para programação por objetos. Cardelli e Wegner apresentam um outro artigo importante para acompanhar o trabalho formal no mundo dos objetos [CW 85].

O trabalho de formalização tem um papel mais amplo do que fornecer um respaldo teórico a novas idéias, pois introduz formas para especificação dos conceitos relacionados a objetos que, não tendo compromisso com questões relacionadas à implementação em linguagens de programação, conseguem captar de maneira consistente a essência de descrição e utilização de entidades como tipos, operações, objetos, encapsulamento e herança.

3.2.1 Teorias e tipos

Uma vez que nosso objetivo com a formalização é sedimentar uma visão abstrata do paradigma antes de passarmos a trabalhar com construções de linguagens abordaremos apenas teorias que consideramos formas naturais para especificação e manipulação de abstrações. Podem ser encontradas na literatura modelagens para tipos que envolvem entidades matemáticas bem sofisticadas (e interessantes), mas neste trabalho apresentamos apenas idéias simples com bom poder de expressão.

Uma abordagem através de álgebra de primeira ordem

A utilização de idéias algébricas como apoio teórico para questões de programação há algum tempo vem sendo largamente explorada. Alguns trabalhos da última década indicaram possíveis caminhos para conexão das idéias teóricas às práticas. Por exemplo, Goguen em [Go 84] introduz uma especificação de software através de ferramenta algébrica que pode ser uma técnica poderosa para reutilização confiável de software. Ele introduz novas idéias que associam aspectos formais aos módulos (*teorias*, que declaram as propriedades formais dos módulos e interfaces do programa; *visões*, que associam teorias a módulos; *expressões de módulo*, que produzem novos módulos a partir de já existentes) e exemplifica suas idéias através da linguagem de programação OBJ2 [OBJ2] (de “ultra-alto” nível, baseada em regras de reescrita). Veremos a seguir uma maneira de especificar tipos abstratos de dados através da álgebra.

Uma *álgebra* é uma entidade matemática composta por conjuntos de valores e operações que manipulam esses valores. Observe que esta entidade matemática cai como uma luva para representar a noção adotada para tipos na seção 3.1.1.

Uma *assinatura* para uma álgebra A é uma declaração dos conjuntos, constantes e operações de interesse. Assim, uma assinatura $\Sigma = (S, F)$ consiste de um conjunto S de nomes (*sorts*) e um conjunto F de símbolos de operações. Para cada operação f em F temos uma declaração de f relacionando os conjuntos envolvidos na operação

$$f : s_1 s_2 \dots s_k \rightarrow s_{k+1} \quad s_i \in S$$

Uma álgebra sobre uma assinatura (S, F) é uma família $(s^A)_{s \in S}$ de conjuntos (*carrier sets*) e uma família de funções parciais $(f^A)_{f \in F}$ tais que as definições de f^A coincidem com as de $f \in F$.

Por exemplo,

$$S = \{s_1, s_2, s_3\}$$

$$F = \{f_1 : s_1 \times s_2 \rightarrow s_2, \quad f_2 : s_1 \rightarrow s_2, \quad f_3 : s_1 \rightarrow s_3\}$$

definem uma assinatura.

Uma álgebra $A_1(\Sigma)$ pode ser definida associando

$$\{0, 1\} \text{ a } s_1 \text{ e } s_2 \text{ e } \{(0,0), (1,1)\} \text{ a } s_3$$

e

$$(i, j) \rightarrow i \star j \text{ a } f_1, \quad i \rightarrow i \text{ a } f_2, \quad i \rightarrow (i, i) \text{ a } f_3.$$

A álgebra $A_1(\Sigma)$ é um tipo (conjunto de valores $\{0, 1\}$ dotado de duas operações unárias e uma operação binária). A assinatura da álgebra pode ser encarada como uma declaração que especifica todos os elementos envolvidos na abstração.

Vejamos um outro exemplo de assinatura:

```
num, bool:      tipo;
zero, um:       função() que devolve num;
+, *:          função(num, num) que devolve num;
verdadeiro, falso: função() que devolve bool;
e, ou:         função(bool, bool) que devolve bool;
é-par, é-ímpar: função (num) que devolve bool;
```

A assinatura introduz os nomes de interesse, mas não atribui significado algum a esses nomes, revelando apenas que duas espécies de origens para valores, três tipos de operações e algumas constantes (definidas através de funções sem parâmetros) serão

utilizadas. Para associarmos uma semântica à assinatura, ou seja definirmos uma álgebra, basta especificarmos os conjuntos e ligar as operações com funções.

Considere a definição abaixo de uma álgebra que chamaremos de Mod2:

- Assinatura para Mod2

número: *tipo*;
 $+$, $*$: *função*(número, número) *que devolve* número;

- Semântica para Mod2

número == {0, 1}
zero() == 0
um() == 1
 $+(x, y)$ == *se* $x = 0$
 então y
 senão se $y = 0$
 então x
 senão 0
 $*(x, y)$ == *se* $x = 0$ *ou* $y = 0$
 então 0
 senão 1

Note que Mod2 representa satisfatoriamente a aritmética módulo 2, mas não pode ser associada diretamente a uma unidade que encapsule o tratamento de tais números pois não há na álgebra conceito de estado ou armazenamento.

Os que têm o ponto de vista de que o importante em um tipo abstrato de dados é a abstração e não o estado de armazenamento encontram em álgebras uma conceituação teórica que capta a essência dos tipos abstratos. Uma vez presentes tais especificações para os tipos no software caberia ao sistema tratar os objetos (valores) de maneira consistente com as definições da álgebra, consultando as definições do tipo abstrato a fim de manipular a área de armazenamento associada aos valores.

Para os que consideram que uma formalização matemática “útil” para tipos abstratos de dados deve incorporar o conceito de estado (como uma máquina abstrata) existem alternativas que incluem estados no modelo algébrico para tipos. Uma possibilidade é acrescentar à álgebra subjacente ao tipo abstrato constantes extras que podem ser usadas para armazenar um número. Quando quisermos alterar o valor “basta” alterar a álgebra de maneira que a constante defina o novo valor desejado. A noção de estado pode ser muito importante para algumas áreas de aplicação como simulação.

Como é utilizada álgebra de primeira ordem as funções e a própria álgebra não são valores de primeira classe, ou seja, não podem ser utilizados da mesma maneira

que as constantes e os valores dos conjuntos de suporte da álgebra. Isto pode ser bem restritivo, pois manipular o próprio tipo abstrato durante a execução (obtendo novas unidades) parece ser útil se pensamos em ambientes de programação onde unidades já definidas poderiam ser utilizadas para construção de novas. Em outras palavras, fica difícil formalizar herança.

Uma outra aplicação de álgebra e assinaturas para especificação de software pode ser vista em [G-M 88].

Enfoques baseados em tipos existenciais

A motivação central para se trabalhar com estruturas de segunda ordem é expressar o *tipo* de um tipo abstrato de dados. No enfoque algébrico os tipos (*sorts*) ficavam num plano diferente dos tipos abstratos de dados (álgebras) e portanto não poderiam servir de argumento ou valor de retorno de funções. Se permitimos especificação de tipos para a álgebra torna-se possível usar tipos abstratos como parâmetros e, em consequência, valores podem ser examinados em tempo de execução a fim de que uma abstração apropriada fosse escolhida e fornecida às funções que manipulam o valor. Isto pode parecer obscuro se olhamos um valor já atrelando-o ao tipo abstrato de dados que o representa; nesse caso não faz sentido uma função levar em consideração a abstração corrente, já que as próprias funções que manipulam o valor estariam amarradas ao tipo abstrato de dados. Mas um conjunto de valores pode ser abstraído de várias formas. Considere um grafo, por exemplo. São muitas as maneiras de abstraí-lo e algumas funções atuam de maneira mais natural em cima de uma particular estrutura; seria então conveniente que em tempo de execução a abstração fosse escolhida e enviada à função.

O que são tipos existenciais

Um tipo quantificado existencialmente pode ser denotado da seguinte forma:

$$\exists a. t(a)$$

Sendo p um valor e $t(a)$ uma expressão com tipos que contém ocorrências livres de a então $p : \exists a. t(a)$ significa que existe um tipo a para o qual p tem o tipo $t(a)$. Assim, para cada $\tau(a)$ a expressão $\exists a. \tau(a)$ denota um tipo, ou seja um conjunto de valores com comportamento correlato.

Por exemplo:

- $(3,4)$: $\exists a. a \times a$, pois $(3,4)$ tem o tipo $a \times a$ para a assumindo *integer*
- $(3,4)$: $\exists a. a$, pois $(3,4)$ tem o tipo a para a assumindo *integer* $\times$ *integer*

Observe que um valor pode pertencer a vários tipos existenciais distintos. Como qualquer valor v tem o tipo $\exists a. a$ temos que $\exists a. a$ é o tipo mais geral possível, composto de todos os valores do universo.

A quantificação existencial é usada para expressar o *tipo* de um tipo abstrato de dados τ dotado de operações $f_1, \dots, f_n$ cujas definições (de domínio e contra-domínio) são dadas por $\sigma_1, \dots, \sigma_n$:

$$\exists \tau. \sigma_1 \text{ e } \sigma_2 \text{ e } \dots \text{ e } \sigma_n$$

O tipo de `Mod2` é portanto $\exists \alpha. \alpha \times \alpha \rightarrow \alpha$ e $\alpha \times \alpha \rightarrow \alpha$.

Tipos existenciais de Mitchell e Plotkin

O trabalho de Mitchell e Plotkin [MiPl85] usa uma versão estendida do λ -cálculo de segunda ordem chamada SOL para representar tipos abstratos de dados. A linguagem SOL permite que a definição da álgebra seja associada a um tipo.

A forma de descrição segue o seguinte padrão:

assinatura
álgebra
Corpo que utiliza a definição

Vejamos como exemplo de tipo abstrato de dados o conjunto de pontos no $\mathcal{R}^2$, com as operações de adição e obtenção de valores de abscissa e ordenada. A descrição em SOL poderia ser feita da seguinte forma:

```

abstype ponto with
  cria: function(real, real) result ponto;
  soma: function(ponto, ponto) result ponto;
  abs-x: function(ponto) result real;
  ord-y: function(ponto) result real;

is
representation real × real
  cria(x, y) == cons(x, y)
  soma(c1, c2) == cons(car(c1)+car(c2), cdr(c1)+cdr(c2));
  abs-x(c) == car(c);
  ord-y(c) == cdr(c);

in
  ponto.abs-x(ponto.soma(ponto.cria(1,4), ponto.cria(2,3)));

```

A expressão acima foi construída com uma especificação de interface, uma expressão para a álgebra do tipo abstrato e um corpo para o qual as duas definições anteriores são visíveis. No exemplo o corpo é composto de apenas uma expressão cuja avaliação resulta no real 3. A especificação da interface enumera as possíveis manipulações para o tipo abstrato, indicando a existência de um tipo com o nome *ponto* munido de operações *cria*, *soma*, *abs-x*, *ord-y*. Após a interface temos definida a álgebra associada ao tipo, com o conjunto $real \times real$ como base da álgebra e as funções definidas através das operações *cons*, *car* e *cdr* sobre listas (manipulando o par que representa o ponto). Em SOL uma expressão algébrica tem a forma

$$\text{representation } \tau \quad M_1 \dots M_n$$

onde τ é uma expressão envolvendo tipos (conjuntos bases da álgebra) e $M_1 \dots M_n$ são definições de funções.

O tipo da álgebra é expresso por $\exists \tau. \sigma_1 \text{ e } \dots \text{ e } \sigma_n$, onde σ_i é a definição de M_i .

Uma possível definição do tipo Mod2 em SOL é dada por:


```
abstype Mod2 with
  cria: function(inteiro) result Mod2;
  soma: function(Mod2, Mod2) result Mod2;
  mult: function(Mod2, Mod2) result Mod2;
  zero: function() result Mod2;
  um: function() result Mod2;
```

is

representation integer

```
  cria(x)      == x mod 2;
  soma(c1, c2) == (c1 + c2) mod 2;
  mult(c1, c2) == (c1 * c2) mod 2;
  zero         == 0;
  um           == 1;
```

in

```
  Mod2.soma(Mod2.um(), Mod2.zero());
```

Tipos existenciais de Cardelli e Wegner

Cardelli e Wegner projetaram uma linguagem similar a SOL consideravelmente mais flexível: FUN, que fornece meios para especificação e uso da visão matemática de tipos abstratos de dados de uma maneira bem geral.

Os objetos manipulados por FUN não são valores simplesmente já que englobam informação acerca de manipulação em sua estrutura. $\langle 0, +, \star \rangle$ e $\langle 1, +, \star \rangle$ são objetos da álgebra Mod2 sob a visão da FUN e podem ser considerados também, por si só, uma álgebra. O valor (inteiro) 0 pode ser isolado no objeto $\langle 0, +, \star \rangle$ apenas se este objeto for dotado de uma operação para este fim: $\langle 0, +, \star, \text{devolve-valor-int} \rangle$.

A sintaxe da linguagem FUN permite que se declare um valor como tendo um tipo existencial que serve de descritor do tipo abstrato.

Para declararmos $\langle 3, inc \rangle$ como sendo do tipo $\exists \tau. \langle val : \tau, op : \tau \rightarrow integer \rangle$ fazemos

$obj = \text{pack}[\tau = integer \text{ in } \langle val : \tau; op : \tau \rightarrow integer \rangle] \langle 3, inc \rangle$

A definição de um tipo abstrato de dados em FUN é composta pela declaração do tipo abstrato, da assinatura da álgebra (um pouco diferente do que vinha sendo assinatura até agora, pois estaremos apenas especificando a estrutura — *record* — dos valores) e a álgebra.

O tipo abstrato de dados Mod2 ficaria, por exemplo:

```
type Mod2                =  $\exists repr. \text{op-mod2} [ repr ]$ 

type op-mod2              = <
                           zero: function() result repr;
                           um:  function() result repr;
                           soma: function(repr, repr) result repr;
                           mult: function(repr, repr) result repr;
                           >

value repr-num: Mod2 =
  pack [ repr = integer in op-mod2 [repr] ]
    <
      zero = fun() 0;
      um   = fun() 1;
      soma = fun(x,y) x+y mod 2;
      mult = fun(x,y) x * y mod 2;
    >
```

O tipo abstrato tem tipo especificado por uma fórmula existencial com *repr* como parâmetro. Assim, a especificação de *repr*'s distintas permitirá que os valores de um mesmo tipo abstrato sejam manipulados de maneiras distintas internamente, obedecendo — é lógico — à mesma interface.

A expressão *opmod-2* que especifica as operações é definida através de uma estrutura de *record* composta pelo elenco de manipulações do tipo abstrato.

O valor *repr-num* é um tipo de dados que instancia a álgebra (representação). Observe que os tipos abstratos de dados declarados podem ser usados como valores, são objetos como quaisquer outros. Outra representação para *Mod2* poderia ser instanciada com

```
value repr-bool: Mod2 = pack [ repr = boolean in op-mod2 [repr]
  <
    zero  = fun() false;
    um    = fun() true;
    soma  = fun(x,y) x or y;
    mult  = fun(x,y) x and y;
  >
```

Para obtermos outra representação para *Mod2* em SOL poderíamos criar uma outra abstração *Outro-Mod2* que teria o mesmo tipo que *Mod2*, mas essa equivalência de abstração não fica explícita nas especificações.

A definição para o tipo abstrato de dados referente a pontos no plano seria:

```
type ponto =  $\exists$ p-rep. ponto-op[p-rep]
```

```
type ponto-op[p-rep] = <
  cria:  function(real, real) result p-rep;
  soma:  function(p-rep, p-rep) result p-rep;
  val-x: function(p-rep) result real;
  val-y: function(p-rep) result real;
>
```

```
value ponto-cart : ponto =
  pack [p-rep = (real  $\times$  real) in ponto-op[p-rep]]
  <
    cria  = fun(x, y) cons(x,y);
    soma  = fun(p1, p2) cons(car(p1)+car(p2), cdr(p1)+cdr(p2))
    val-x = fun(p) car(p);
    val-y = fun(p) cdr(p);
  >
```


Uma expressão de uso do tipo definido seria:

value resultado = val-x (soma(cria (2,1), cria(2,2))), cuja avaliação resultaria no real 4.

FUN permite que a noção de estado seja inserida na definição da álgebra, colocando a definição do tipo que representaria o estado na estrutura *record* que define a assinatura da álgebra. A evocação do valor de um estado é feita através de uma referência *self* e o estado passa a ser uma espécie de parâmetro implícito das manipulações do tipo abstrato. Construção semelhante aparece nas linguagens Smalltalk e C++ e será abordada no capítulo 4.

3.2.2 Teorias e encapsulamento

A palavra *pack* usada na definição da álgebra sugere *information hiding*, isto é, encapsulamento das operações que restringe o acesso aos valores do tipo abstrato. A forma geral para a operação *pack* é:

$$\text{pack} [a = \text{representação in interface}] < \text{conteúdo} >$$

pack é o único mecanismo para criar objetos de um tipo existencial. Assim, se nós definimos uma variável através de uma declaração como

$$p : \exists a. a \times a \rightarrow a$$

então a única maneira de criar valores com o tipo *p* é através da operação *pack*.

Nos exemplos que fizemos com os tipos abstratos nós exemplificamos o uso de objetos através de chamadas de funções, como por exemplo em

$$\text{val-x (soma(cria (2,1), cria(2,2)))}$$

Na verdade a manipulação de objetos deve ser feita de maneira mais criteriosa (que simplifica a teoria de FUN para checagem de tipos). Todo objeto a ser manipulado deve ser criado através de *pack* e aberto para uso através de *open*. A operação *open* deve ser usada da seguinte forma:

$$\text{open obj as id[t] in } <\text{expressão}>$$

Assim, a manipulação correta para pontos (respeita o encapsulamento da representação) deve ser feita da seguinte forma:

```
open ponto-cart as p in p.val-x (p.soma( p.cria (1,1), p.cria(2,2)))
```

O identificador *p* é visível somente dentro do escopo definido pelo *open*. Mas veja como a utilização da entidade “aberta” tem algumas restrições sérias, por exemplo a manipulação abaixo é ilegal:

```
open ponto-car as p1 in
  open ponto-car as p2 in
    p1.val-x(p1.soma( p1.cria (1,1), p2.cria(2,2)))
```

A expressão acima cria dois identificadores associados ao objeto ponto-car, *p1* e *p2*. O problema é que *p1.soma* pode ser aplicado apenas a pontos criados pela operação *p1.cria*, o sistema de checagem de tipos não detecta equivalência entre *p1* e *p2* e portanto rejeita a chamada *p1.soma* com operandos não associados a *p1*. Uma mudança simples no sistema de checagem de tipos faria com que os objetos *p1* e *p2* pudessem ser utilizados indistintamente.

3.2.3 Teorias e herança

Na seção 3.1.5 nós discutiremos sobre herança em duas linhas: mecanismo de relacionar tipos (e portanto comportamento) e mecanismo de construção de unidades de software a partir de peças já existentes.

Esboçar uma formalização para derivação de unidades envolveria construtores para adicionar, modificar ou extinguir operações do módulo base. Não seria difícil inculcar tal mecanismo na linguagem FUN, por exemplo. Como resultado teríamos coleções de tipos abstratos diferentes, relacionados por derivação, e nenhuma informação extra sobre como se relacionam os conjuntos de valores de cada um dos tipos abstratos. Definindo um construtor para herança que considere válido herdar apenas parte de uma especificação (modificando ou eliminando o restante) impossibilitamos que valores sob responsabilidade de um tipo abstrato derivado seja utilizado pelas operações da abstração original, pois não haveria maneira de garantir que há consistência de comportamento entre o tipo que dá e o que recebe através de herança.

Especificaremos o mecanismo de herança de maneira que fique definida a relação entre os tipos subjacentes aos tipos abstratos de dados. Isso implicará em que o uso de herança com intuito de reutilização de software seja um pouco restrito. A reutilização deverá expressar reuso de especificação ao invés de um fatoramento de rotinas comuns.

Através de herança desejamos obter um novo tipo abstrato de dados a partir de outro já especificado. Nesse sentido, buscamos então um construtor para tipos abstratos. A noção de construtor é usual para caracterização de novos conjuntos:

tendo t_1 e t_2 já definidos um novo conjunto pode ser obtido pelos construtores $\times$ e $\rightarrow$, como por exemplo $(real \times real)$ e $(integer \rightarrow real)$.

Noção de herança em FUN

Consideremos o tipo abstrato de dados que manipula pares de inteiros, definido a seguir:

```
type def-par =  $\exists$ p-rep. op-par[p-rep]

type op-par[p-rep] = <
    cria:  function(integer, integer) result p-rep;
    prim:  function(p-rep) result integer;
    seg:   function(p-rep) result integer;
>

value par : def-par =
    pack [p-rep = (integer  $\times$  integer) in op-par[p-rep]]
    <
        cria  = fun(x, y) cons(x,y);
        prim  = fun(p) car(p);
        seg   = fun(p) cdr(p);
    >
```

Observe que a definição de objetos é feita através de *records* como:

```
<
    cria:  function(integer, integer) result p-rep;
    prim:  function(p-rep) result integer;
    ult:   function(p-rep) result integer;
>
```

Um *record* é uma associação (finita) de rótulos e valores. A expressão *rótulo* = *valor* denota um campo. Por exemplo $\langle \text{nome} = \text{Claudio}; \text{idade} = 16 \rangle$ tem dois campos e a ordem destes não é importante, pois cada valor é acessado exclusivamente através de seu rótulo.

O tipo de um record é uma caracterização para cada um de seus rótulos, de maneira que o valor associado a um rótulo seja do tipo que o define. Observe que na definição de um tipo abstrato de dados em FUN records são utilizados para denotar a lista de operações disponíveis.

A definição de tipos para records são tipos em FUN (por exemplo <dia: 1..31; mês: 1..31>). Há um construtor para concatenação desses tipo-records:

$$\{r_1 : t_1; r_2 : t_2; \dots; r_k : t_k\} \& \{r_{k+1} : t_{k+1}; r_{k+2} : t_{k+2}; \dots; r_n : t_n\} = \{r_1 : t_1; \dots; r_n : t_n\}$$

Definimos a relação de subtipo ($\leq$) entre tipo-records da seguinte maneira:

$$\{a_1 : t_1; \dots; a_k : t_k; \dots; a_n : t_n\} \leq \{a_1 : u_1; \dots; a_k : u_k\}$$

se e somente se $t_i \leq u_i$ para $i \in \{1, \dots, k\}$

Em outras palavras um tipo-record A é um subtipo de outro tipo-record B se A tem todos os campos de B (e possivelmente mais) e os tipos dos campos em comum obedecem também a relação de subtipo (para tipos não estruturados a relação de subtipo equivale a $\subseteq$). É através da idéia de subtipo que o conceito de herança é expresso em FUN. O operador $\&$ serve de construtor de novos tipos, sempre conservando as operações para cada um dos tipos presentes nos campos.

Acompanhemos um exemplo de subtipo para perceber como a “doação” de comportamento é feita. Vamos especificar o tipo abstrato outro-par, que representará pares de inteiros semelhantes aos definidos em par que possuem mais uma operação, *inverte*, cujo efeito é devolver um novo par com os componentes do par original trocados. A definição pode ser feita da seguinte maneira:

```
type op-outro-par = op-par[par] & < inverte: function(par) result par >;
```

```
value outro-par : op-outro-par =
  pack [op-outro-par]
  <
    inverte = fun(p) cons(seg(p), prim(p));
  >
```

Para que outro-par herdasse as especificações feitas em par foi usada concatenação de records, o que no fundo significa compor o comportamento de outro-par a partir do definido para par. Se utilizamos um objeto do tipo outro-par através de

```
open outro-par as p in p.inverte(p.create(1,2))
```

como $\text{op-outro-par} \leq \text{op-par}$ é natural então que um objeto do tipo outro-par seja também do tipo par, já que engloba o comportamento. Assim, $\text{p.create}(1,2)$ é uma chamada válida, mesmo p sendo um identificador relacionado ao tipo abstrato outro-par e *create* um operador definido para o tipo abstrato par. A relação de ordem entre os *record-type* expressa uma relação entre os comportamentos e o mecanismo de herança fica sendo então uma maneira de concatená-los; não é possível redefinição de comportamento (alteração de operações herdadas). Observe que como outro-par foi definido a partir de par ele delega a um objeto do tipo par toda a manipulação envolvida nas operações *cria*, *prim* e *seg*.

A formalização de tipos através de *records*, que discutiremos na seção 3.2.4, apresenta a vantagem de se basear numa estrutura usualmente utilizada para implementação de classes em linguagens de programação. Wegner em [Weg 88] explora em detalhe a utilização de *records* para modelar tipos apresentando herança como um mecanismo de modificação incremental. Modificação incremental é uma idéia genérica que facilita reutilização, aproveitando conceitos ou dispositivos físicos para a construção de algo similar. Herança é um tipo particular de mecanismo que transforma uma entidade P através de um modificador M obtendo como resultando a entidade $R = P + M$:

O trabalho de Wegner caracteriza as propriedades de um modificador e correspondentes relações entre as entidades envolvidas na herança. Na seção 3.5 voltamos a abordar a utilização de records como tipos.

Dois trabalhos independentes ([Cook89],[Red 88]), introduziram recentemente uma nova modelagem para herança que, segundo os autores, foi originalmente desenvolvida para modelar o conceito em linguagens orientadas por objetos mas que acabou mostrando que herança é um mecanismo geral aplicável a qualquer tipo de definição recursiva. Eles, como Wegner, consideram herança um mecanismo de modificação incremental e acham que uma modelagem para o processo de derivação deve representar a verdadeira modificação do componente original:

O modelo é formalizado através de semântica denotacional.

3.2.4 Herança em OBJ2

A linguagem OBJ2 expressa herança através de um mecanismo de aproveitamento de módulos, que são as unidades que contém a descrição algébrica de um tipo abstrato de dados. Através do exemplo a seguir, que especifica os tipos *lista* e *lista-ordenável*, podemos observar que a criação de tipos abstratos a partir de já existentes também é feita estendendo comportamento já definido. Consideramos o trabalho de Goguen em OBJ2 interessante pela forma de associação de teorias a módulos.

```
module LISTA[Objeto::Qualquer] is
  using NAT.
  sorts Nlista Lista,
  subsorts Nlista < Lista,
  ops
    nil :    → Lista,
    cons: Objeto Lista → Nlista [associativa com identidade=nil],
    prim: Nlista → Objeto,
    resto: Nlista → Lista;
    tam:  Lista → Nat;
  vars
    X:    Objeto,
    L:    Lista.
  axioms
    prim(cons(X, L)) = X.
    resto(cons(X, L)) = L.
    tam(nil) = 0.
    tam(cons(X, L)) = 1 + tam (L).
endmod LISTA

module LISTA-ORDENÁVEL [X :: POSET] is
  using BOOL.
  extending LIST[X]
  ops
    ordenada: Lista → Bool,
    ordena:  Lista → Lista.
    :
endmod LISTA-ORDENÁVEL
```

A linguagem FOOPS[Go 86], também desenvolvida por Goguen, é baseada na notação utilizada por OBJ2; como foi criada com o objetivo de unificar a visão dos objetos com as idéias de programação funcional engloba também um tipo primitivo para programas.

3.2.5 A especificação para herança em GSBL

Clerici e Orejas em [C-O 88] descrevem uma linguagem de especificação algébrica baseada em herança: GSBL. O objetivo desta nova forma de especificação é obter uma formalização para herança mais próxima do uso desta como metodologia de desenvolvimento de programas. Embora reconheçam na relação de subtipo uma noção importante que permite a expressão de uma estrutura não rígida para tipos, consideram-na insuficiente para expressar herança de uma maneira geral. A base do trabalho deles são especificações incompletas e uma nova relação de subclasse, com o sentido de “é mais detalhada que”. Observe a sintaxe GSBL na descrição a seguir, que especifica um conjunto ordenado qualquer:

```

CLASS   With-order
OVER    Boolean
WITH    SORTS With-order
        OPS  $=, \leq$ : With-order  $\times$  With-order  $\rightarrow$  Boolean
        EQS  $\{a, b, c : \textit{With-order}\}$ 
             $a = a = \textit{true}$ 
             $a = b \vee \neg(a = b) = \textit{true}$ 
             $a = b \Rightarrow b = a = \textit{true}$ 
             $(a \leq b) \vee (b \leq a) = \textit{true}$ 
             $(a \leq b \wedge b \leq c) \Rightarrow a \leq c = \textit{true}$ 
             $(a \leq b) \wedge (b \leq a) = (a = b)$ 
        DEFINE OPS  $\geq, <, >$ : With-order  $\times$  With-order  $\rightarrow$  Boolean
            EQS  $\{a, b : \textit{With-order}\}$ 
                 $a \geq b = b \leq a$ 
                 $a > b = \neg(a \leq b)$ 
                 $a > b = b < a$ 
ENDCLASS

```

Uma classe para os números naturais pode ser definida a partir de *With-order*:

```

CLASS      Natural
SUBCLASS-OF With-order
DEFINE     SORTS Natural
           OPS  $=, \leq$ 
               0:  $\rightarrow$  Natural
               succ: Natural  $\rightarrow$  Natural
               0 = succ(a) = false
               succ(a) = succ(b) =  $a = b$ 
               0  $\leq$  succ(a) = true
               succ(a)  $\leq$  succ(b) =  $a \leq b$ 
ENDCLASS

```

Observe que as especificações são feitas através de restrições, obtendo descrições cada vez mais detalhadas. Os autores descrevem em seu trabalho a semântica de cada uma das partes sintáticas da linguagem, descrevendo em detalhe as propriedades matemáticas das especificações. Duas coisas diferenciam esse trabalho dos muitos outros que envolvem especificações: a presença de vários exemplos (alguns não triviais) e a apresentação de uma hierarquia de especificações através de uma figura que propõe notações gráficas para representar as maneiras pelas quais as classes se relacionam.

3.2.6 Tipos, de novo

Na nossa apresentação das formalizações enfatizamos como os diferentes enfoques especificavam e relacionavam tipos e nos deparamos com duas linhas principais para conceituação de um tipo: o trabalho com tipos existenciais, baseados na noção de records, e a especificação algébrica baseada em restrições (caracterização do conjunto via axiomas, como em OBJ2 e GSBL). Mas, independente da abordagem, tentamos frisar durante toda a conceituação a idéia de tipo como *conjunto de valores com mesmo comportamento, descrito através de um conjunto de operações*, conforme apresentamos nas seções iniciais deste capítulo.

Nós propomos a seguir uma formalização matemática para a nossa noção de tipos.

Dados conjuntos A, B a notação $A \subseteq_{\alpha} B$ (leia-se A é α -subconjunto de B) significa que $\alpha : A \rightarrow B$ é uma função injetora; ela será usada para indicar uma forma fixa pela qual cada elemento de A pode ser considerado como membro de B .

Um *tipo* é um par (C, F) tais que:

1. C é o conjunto de valores descrito pelo tipo;
2. F é um conjunto de identificadores $f_1, f_2, \dots, f_n$ onde a cada f_i está associada uma função *parcial* $\varphi_i : d_i \rightarrow c_i$

Dessa forma temos a seguinte definição para *subtipo*:

Dados os tipos $T = (C, F)$ e $T' = (C', F')$, T' é **subtipo** de T ($T' \leq T$) se e somente se:

1. $C' \subseteq_{\alpha} C$
2. $F' \supseteq F$
3. para cada $f'_i \in F'$
 - $d'_i \subseteq_{\alpha_i} d_i$ e $c'_i \subseteq_{\beta_i} c_i$
 - $\forall v \in d'_i \quad \varphi_i(\alpha_i v) = \beta_i(\varphi'_i(d'_i))$

Observações:

- A noção de subtipo envolve todas as injeções α_i, β_i utilizadas para passar de um tipo para outro;
- Em geral, as injeções desejadas são óbvias e podemos simplificar a notação substituindo $\subseteq_\alpha$ por um símbolo de inclusão. Poderíamos apresentar a definição da seguinte forma:

Dados os tipos $T = (C, F)$ e $T' = (C', F')$, T' é **subtipo** de T ($T' \leq T$) se e somente se:

1. $C' \subseteq C$
2. $F' \supseteq F$
3. para cada $f'_i \in F'$ sendo $\varphi'_i : d'_i \rightarrow c'_i$ a função associada a f'_i em T' e $\varphi_i : d_i \rightarrow c_i$ a função associada a f_i em T temos que:
 - * $d'_i \subseteq d_i$ e $c'_i \subseteq c_i$
 - * para todo valor $v \in d'_i$ se $\varphi_i(v)$ e $\varphi'_i(v)$ estão definidos então $\varphi_i(v) = \varphi'_i(v)$

Por exemplo para os tipos:

números reais: $(\mathbb{R}, \{+, -, *, /, \sqrt{\cdot}\})$

números racionais: $(\mathbb{Q}, \{+, -, *, /, \sqrt{\cdot}\})$

números inteiros: $(\mathbb{Z}, \{+, -, *, /, \sqrt{\cdot}, succ, pred\})$

números naturais: $(\mathbb{N}, \{+, -, *, /, \sqrt{\cdot}, succ, pred, \acute{e}\text{-}par, \acute{e}\text{-}impar\})$

temos $\mathbb{N} \leq \mathbb{Z} \leq \mathbb{Q} \leq \mathbb{R}$

Observe que a definição adotada tenta captar que há uma especialização no comportamento. Um elemento do subtipo pode ser usado em qualquer contexto em que um elemento do supertipo é esperado pois as restrições impostas sobre as funções que manipulam os valores garantem consistência de comportamento. Note a forte ligação entre as relações de subtipo e subconjunto.

No trabalho de Wegner[Weg 88] encontramos uma definição para tipos que difere em essência da nossa, já que a relação de subconjunto não induz a de subtipos. Sua conceituação de tipos é baseada nos *records* (conforme vimos na seção 3.2.3), ou seja, um tipo é uma coleção de atributos:

```
type Pessoa1    = (nome: String);  
type Pessoa    = (nome: String; idade: Int(0..120));  
type Aposentado = (nome: String; idade: Int(65..120));
```


A relação de subtipo é inicialmente definida a partir do **princípio da substitutibilidade**: uma instância de um subtipo pode ser usado em qualquer contexto no qual uma instância do supertipo é esperada.

A definição de Wegner, como a nossa, focaliza a consistência de comportamento. Mas ele chama a atenção em seu trabalho ao fato de subconjuntos não serem subtipos, o que entra em conflito com a noção que nós adotamos. Veja o exemplo dado por Wegner para exemplificar sua afirmação:

```
p: Pessoa;  
r: Aposentado;  
p:= r;  
ajusta-idade(p, 40);
```

As duas primeiras linhas declaram os elementos como sendo records com dada estrutura. A chamada `ajusta-idade(p, 40)` falharia já que não é possível que um elemento do tipo `Aposentado` tenha o campo *idade* armazenando um valor inferior a 65. Assim, os tipos `Aposentado` e `Pessoa` apresentam uma relação de subconjunto mas não de subtipo, já que um `Aposentado` não pode ser utilizado em qualquer expressão envolvendo um elemento do tipo `Pessoa`.

A grande diferença entre as definições de tipos através de um par (C, F) e através de *records* não se encontra na definição de subtipo em si, mas sim no fato de que com a definição envolvendo *records* estamos associando a tipo a noção de estado e dando uma identidade a cada valor a partir de sua área de armazenamento. Da forma que modelamos as manipulações de valores de um tipo através de funções que *devolvem* outros valores não há como definir coisas como `Ajusta-idade`, já que não há como transformar um valor, podemos simplesmente associá-lo a outro elemento do mesmo conjunto.

Envolver a noção de estado e funções que alteram seu argumento em uma conceitualização para objetos é uma questão, no momento, muito delicada e polêmica. Para os que vêm na formalização uma oportunidade de conseguir um distanciamento de máquinas e implementações propício para que sejam fixadas as idéias centrais do paradigma é mais conveniente trabalhar com base apenas em conjuntos de valores, tentando descobrir uma visão de tipos e herança razoável tanto para desenvolvimento *top-down* (lidando com subtipos) como *bottom-up* (que envolve combinação de classes).

O trabalho de Horn [Hor 87] apresenta uma relação entre tipos abstratos de dados chamada *conformance*, que é um pouco mais geral (no sentido de formas possíveis para derivação de classes) do que a relação de subtipo tentando captar um equivalência de uso entre objetos, ou seja, se $T_1 \leq T_2$ então:

1. Uma expressão do tipo T_1 pode ser atribuída a um objeto do tipo T_2 ;

2. Um argumento do tipo T_1 pode ser enviado a função com parâmetro formal do tipo T_2 ;
3. Um resultado de função do tipo T_1 pode ser atribuído a uma expressão do tipo T_2 .

Um excelente trabalho sobre formas adequadas para modelar tipos é desenvolvido por Wegner em [Weg 86], que explora em detalhe a idéia de classificação em sistemas orientados para objetos com um enfoque filosófico. Em [BCW 86] Wegner apresenta detalhadamente um modelo algébrico para subtipos.

Uma vez que as conceituações existentes para herança se adequam mais a derivações que retratem subtipo do que reutilização de unidades (onde o comportamento da superclasse pode ser arbitrariamente modificado) consideramos conveniente que no uso efetivo do paradigma (ou mesmo em uma teoria para derivação de tipos) existam mecanismos distintos para as duas espécies de derivação.

3.3 Polimorfismo

A maior parte do estudo feito nesse capítulo é dedicada a tipos: como especificá-los ou relacioná-los. Os valores sempre estavam associados a algum tipo, na verdade um valor como $(0, 1)$ por exemplo assumiu durante nossos exemplos tipos distintos. Assim, trabalhamos o tempo todo considerando o aspecto *polimórfico* dos valores. Como linguagens de programação convencionais como Pascal são baseadas na idéia de que variáveis, funções e procedimentos têm um único tipo o conceito de *polimorfismo* em linguagens de programação de maneira geral merece atenção especial.

Linguagens polimórficas são aquelas em que alguns valores ou variáveis podem ter mais de um tipo. Funções polimórficas, também chamadas genéricas, são funções cujos operandos (parâmetros formais) podem ser de mais de um tipo. Tipos polimórficos podem ser definidos como tipos cujas operações são aplicáveis a operandos de vários tipos.

No trabalho de Cardelli e Wegner [CW 85] podemos encontrar uma classificação envolvendo vários tipos de polimorfismo desenvolvida de maneira a englobar subtipos e herança, baseada numa divisão clássica para tipos de polimorfismo: polimorfismo *paramétrico* e polimorfismo *ad-hoc*. São indicadas as seguintes categorias de polimorfismo:

As diversas variedades tentam captar a maneira em que a generalidade (de funções, em particular) é obtida e utilizada:

1. *polimorfismo universal*

É o polimorfismo onde as funções trabalham uniformemente sobre uma coleção de tipos, que em geral possuem alguma estrutura comum. Essa uniformidade pode ser atingida de duas maneiras:

- *polimorfismo paramétrico*

É o tipo de polimorfismo que atinge a uniformidade através da passagem de tipos como parâmetros para funções.

- *polimorfismo de inclusão*

As funções que possuem polimorfismo universal através de inclusão são as que trabalham uniformemente em cima de coleções de tipos não disjuntos. Nesse caso a relação de estrutura entre os tipos é a de subtipo. Em Pascal esse tipo de polimorfismo em função aparece, por exemplo, quando declaramos uma função como tendo parâmetros do tipo inteiro e na chamada fornecemos como argumento um identificador com tipo *intervalo*.

As duas idéias acima poderiam ser expressas apenas usando polimorfismo paramétrico, mas esta separação de conceitos facilita a ligação de polimorfismo e herança.

2. *polimorfismo ad-hoc*

As funções manipulam, pelo menos aparentemente, vários tipos diferentes (com estrutura nada correlatas, às vezes) de maneira distinta para cada tipo. Os mecanismos mais comuns para obtenção dessa generalidade aparente são:

- *sobrecarga*

Um mesmo identificador é utilizado para denotar diferentes funções; o contexto de uso da função indicará qual função está sendo referenciada. Como um pré-processamento das chamadas das funções poderia eliminar a sobrecarga de nomes fornecendo novos identificadores a cada uma das diferentes conotações da função, o polimorfismo baseado em *sobrecarga* parece ser uma ilusão sintática (bem útil) de trabalho com funções genéricas. O operador +, por exemplo, é usualmente utilizado para denotar funções de diferentes tipos: soma de inteiros, soma de reais, concatenação de strings.

- *coerção*

Nas funções que tratam diferentes tipos através de coerção existe uma conversão de argumentos para os tipos esperados pelas funções. É o que acontece em Pascal quando definimos uma função com parâmetro real e a avaliamos com um argumento inteiro. A idéia de coerção está presente também na linguagem C, onde a conversão de valores é feita por *casting*.

No caso do polimorfismo universal podemos realmente dizer que as funções têm muitos tipos, mas no caso do polimorfismo ad-hoc podemos ter em última análise apenas um pequeno conjunto de funções monomórficas. Em termos de implementação uma função universalmente polimórfica irá executar o mesmo código para todos os tipos admissíveis como parâmetro enquanto que uma função de polimorfismo ad-hoc pode executar rotinas diferentes para cada tipo de argumento.

Polimorfismo paramétrico é a forma mais pura de polimorfismo: o mesmo objeto ou função pode ser usada uniformemente em diferentes contextos de tipos sem mudança, coerções ou testes especiais feitos em tempo de execução para detecção do tipo. Mas a uniformidade em relação a tipos pode ter um custo alto pois exige uma certa uniformidade de estrutura de dados ou de manipulação da mesma, em geral feita através de apontadores para representação.

Como veremos no capítulo 5, uma das grandes vantagens da programação por objetos é a possibilidade de especificação de polimorfismos de uma maneira natural. As diversas linguagens acabam restringindo as formas de polimorfismo possíveis, mas só o fato de conseguirmos de maneira “limpa” a sobrecarga de identificadores já é um grande passo em direção à uma real utilização da idéia de atrelamento de estruturas de dados às funções que a manipulam.

O conceito de polimorfismo pode ser encaixado de maneira simples no esquema de formalização para tipos abstratos de dados estudado na seção 3.2.1. Funções de tipo genérico são descritas através de **tipos universais**:

$$\forall a. t_1(a) \rightarrow t_2(a)$$

A expressão acima denota todos os tipos de funções que levam elementos do tipo $t_1(a)$ em $t_2(a)$. Por exemplo, $\forall a. a \rightarrow a$ representa um tipo de função. Que tipo de valor é denotado por esse tipo? A função soma1: $(x) \rightarrow (x+1)$ claramente não é desse tipo, já que só faz sentido para elementos x que possam ser manipulados pelo operador de adição. A função identidade $\text{id}: x \rightarrow x$ se aplica para qualquer tipo adotado para x , logo id é do tipo $\forall a. a \rightarrow a$. O tipo e uma função com esse tipo podem ser declarados em FUN da seguinte maneira:

`type tipo-em-tipo =  $\forall a. a \rightarrow a$`

`id: tipo-em-tipo`

Já sabemos que `id` é a função identidade, mas como usá-la e como definir seu corpo? Definir `id` na verdade significa especificar um número infinito de funções:

```
id: fun(x: int) x
id: fun(x: bool) x
id: fun(x: real) x
id: fun(x: ponto) x
id: fun(x: Mod2) x
⋮
```

Isso pode ser feito em FUN através de

```
id: all [t] fun(x : t) x
```

e assim `id[int](3)` devolve como resultado o `int 3`.

Temos assim polimorfismo paramétrico, onde a ação da função é uniforme sobre uma classe de tipos.

O trabalho de Ohori e Buneman propõe um sistema de tipos que engloba a definição de classes com interface paramétricas[O-B 89].

3.4 Herança múltipla

Herança múltipla ocorre quando um objeto pertence a várias classes incomparáveis, ou seja, possui mais de um antecessor imediato na hierarquia de classes. Por exemplo:

Na seção 3.1.5 vimos que este tipo de hierarquia pode ser usado para modelar intersecção de conjuntos de valores. Sob o aspecto de reutilização de unidades herança múltipla nos parece muito importante, já que poderia ser usada como mecanismo de combinação de unidades.

Em termos de conceituação formal, herança múltipla não é por si uma novidade e pode ser retratada pelas teorias que lidam com herança (pode ser especificada em OBJ2, por exemplo). Cardelli julga conveniente apresentar uma semântica “limpa” a fim de salientar quais são os pontos fundamentais ao conceito e quais são otimizações e detalhes de implementação. Em [Car 84] ele introduz uma linguagem, sua semântica, um sistema de tipos e um algoritmo para checagem de tipos.

Há aspectos delicados na idéia de herança múltipla, como o confronto de nomes entre as operações herdadas, que veremos no próximo capítulo.

Capítulo 4

Linguagens e Objetos

Discutiremos neste capítulo algumas formas existentes para expressar as idéias do paradigma dos objetos em uma linguagem de programação. Pretendemos fornecer um pequeno panorama das *linguagens de programação orientadas a objetos* mais em evidência no momento, discutindo alguns pontos interessantes de sua implementação. Nossa intenção é abordar construções sintáticas apenas quando estas são imprescindíveis para o entendimento da concepção adotada pela linguagem acerca de objetos, classes e herança. Concluimos o capítulo tecendo comentários sobre a questão da implementação da herança múltipla, pois muitos trabalhos têm se dedicado a este problema recentemente.

4.1 Linguagens

Discernir se uma linguagem realmente *suporta* programação por objetos não é uma tarefa simples, pois não há uma visão uniforme do que é esperado de uma tal linguagem. Como vimos no capítulo 2, filosofias distintas motivaram as diversas linhas de desenvolvimento de linguagens e ambientes; como resultado os adeptos de uma linguagem enxergam nela os mecanismos necessários para expressar sua visão própria do paradigma dos objetos.

As diversas linhas de conceituação para o paradigma geram necessidades distintas em termos de linguagens:

- Os que consideram que programação por objetos deve ter como base a idéia de “mundo de objetos que se comunicam através de mensagens” precisam de um ambiente onde todos os componentes exerçam papel semelhante. Não seria possível portanto trabalhar com linguagens tradicionais como C e Pascal, onde as várias estruturas têm características distintas. Se tudo é objeto e toda ação é requisitada através de mensagem deve haver um protocolo único para ativação

de rotinas e manipulação de memória.

- A linha que vê em programação por objetos uma maneira de aproximar o modelo de computação do modelo físico de atributos e ações sobre esses atributos privilegia a *simulação* como área de aplicação. Dessa maneira se torna importante que a linguagem forneça mecanismos para processos concorrentes e alternantes, aliados a boa capacidade de abstração.
- Muitos consideram reutilização de tipos abstratos de dados o cerne de programação por objetos. Há os que consideram, porém, que esse princípio não é nenhuma novidade em si, podendo ser seguido através de simples disciplina no projeto de sistemas, sem necessidade de novos ambientes ou ferramentas.
- Os que consideram que o paradigma é forte por propor reutilização de especificação — ao invés de reaproveitamento de código — mantendo um modelo de máquina apropriado para processamento eficiente, têm necessidade de ambientes voltados para encapsulamento e derivação de tipos.

Enfim, no âmbito das linguagens de programação por objetos o certo ar nebuloso que ainda envolve o paradigma persiste: nota-se que os objetivos são correlatos, mas não há uma visão uniforme para implementação dos pontos centrais ao paradigma.

A essência das conceituações formais é o atrelamento de valores às funções que os manipulam, resultando numa unidade que pode ser usada para definir novos valores com funções associadas. Considerando esta noção como uma idéia unificadora para o desenvolvimento de programas baseados em objetos é possível enumerar algumas questões importantes do projeto e implementação de linguagens de programação que efetivamente forneçam suporte ao paradigma:

1. descrição do objeto

O mecanismo para descrição de uma classe de objetos deve permitir que a idéia de tipos abstratos de dados fique explícita na construção sintática para definição de classe. Com base nas informações especificadas em uma classe deve ser possível deduzir que características do objeto são de acesso público (funções, por exemplo) e quais das informações acerca de comportamento são transmitidas para subclasse (estrutura de dados, todas as funções ou parte delas, etc).

2. criação e manipulação de objetos

A linguagem deve fornecer mecanismos para criação (e talvez destruição) de objetos. Os problemas usuais de alocação de área e endereçamento devem ser tratados. O acesso à estrutura de dados deve ser restrito às operações especificadas na classe.

3. especificação de hierarquia

A possibilidade de reaproveitamento de tipos já definidos envolve cuidados para alocação e uso de um novo objeto. Observe que ao criarmos um objeto de uma subclasse estamos ativando uma entidade que acumula o comportamento de todos os seus “ancestrais” na hierarquia, de forma que estaremos na verdade

realizando uma seqüência de operações para alocação. Quanto ao comportamento, quando uma operação é mencionada o código correspondente só poderá ser identificado levando em consideração a classe do valor manipulado a fim de localizar a operação adequada na hierarquia. Acompanhe o exemplo:

Sendo d um objeto associado à classe D , a operação $f(d)$ não será encontrada na interface de D ; f é uma característica de comportamento herdada de A na hierarquia e redefinida em B . A informação disponível em tempo de compilação não é suficiente para encontrar a função desejada, pois poderíamos ter o seguinte caso:

operação FazAlgo (*ObjetoQualquer* o , A a) devolve *ObjetoQualquer*;

```

...
f(a);
...

```

Só em tempo de execução é possível determinar o grau de especialização de a em relação a A e assim identificar a rotina que será atrelada a chamada $f(a)$. Muitos métodos existem para representar internamente a associação entre classes e funções. As linguagens *Smalltalk* e C++ tratam de maneira distinta o problema, exemplificando os métodos mais comuns para encontrar a operação desejada na hierarquia.

Note que a informação de hierarquia de tipos deverá ser utilizada também para verificação de tipos em chamadas de funções e atribuição de valores. No caso de herança múltipla a questão fica ainda mais delicada, pois todo o grafo de hierarquia deverá ser levado em consideração na hora da criação e manipulação de objetos.

Nas seções seguintes discorreremos superficialmente sobre algumas linguagens de programação. As duas vedetes do mundo dos objetos, *Smalltalk* e C++, recebem atenção especial: a primeira por introduzir uma concepção nova para programas e programadores; a segunda por aproveitar a linguagem C (com sua visão padrão para programas, ambientes e máquinas) como base para um trabalho voltado para tipos abstratos de dados e sua reutilização.

4.1.1 C++

C++ é um superconjunto da linguagem C desenvolvido por Bjarne Stroustrup na Bell Laboratories. Foi projetado para:

1. Tornar as construções da linguagem C mais seguras, impondo através de sintaxe formas de utilização que antes ficavam por conta do estilo e disciplina do programador. Foram introduzidas na linguagem, entre outras coisas, checagem e coerção de tipos na passagem de parâmetros, possibilidade de definição de valores *default* para parâmetros não fornecidos à função, declaração de constantes. A declaração de variáveis passou a ser encarada como um comando qualquer da linguagem podendo portanto aparecer no corpo de uma repetição, por exemplo.
2. Permitir a especificação de tipos abstratos de dados.
3. Fornecer um mecanismo para utilização das idéias de programação por objetos, ou seja, reutilização através de herança.

O desenvolvimento da linguagem C++ confrontou-se todo o tempo com a questão de eficiência. Qualquer construção da linguagem ou forma de implementação deveria ser feita de maneira a manter as vantagens da linguagem C: flexibilidade, disponibilidade em muitas máquinas, portabilidade.

Codificar um programa em C++ não é muito diferente de fazê-lo em C, portanto a linguagem pode ser utilizada sem estar associada com programação por objetos, apenas para obtenção de programas onde a manipulação dos tipos usuais é feita de maneira mais segura. Programar por objetos em C++ é mais uma questão de mudança de postura do que de mecanismos ou construções sintáticas: ao invés de analisarmos a funcionalidade do programa a ser desenvolvido procurando rotinas para compor o processamento devemos nos voltar para os dados de interesse utilizados pelo programa a fim de encapsulá-los com as rotinas que os manipulam. Um programa em C++ que trabalhe realmente com as idéias de objetos é resultado de uma espécie de “cooperação”: o programador entra com o estilo de desenvolvimento do programa e a linguagem com mecanismos para que o código reflita as idéias de tipos abstratos de dados e o encapsulamento não fique comprometido. Em contrapartida, quanto mais a linguagem estiver preparada para tratar de maneira eficiente os conceitos de objetos, tipos abstratos de dados e herança mais os programadores se voltarão para este enfoque do desenvolvimento.

A especificação de um tipo abstrato é feita através da declaração de uma classe, ou seja, uma área de armazenamento com uma lista de operações associadas. A visibilidade dos elementos da classe (inclusive das operações) é controlada pela forma de declaração utilizada pelo programador. Existem três possibilidades para definição de acesso: *private*, que restringe a utilização da área ou da operação apenas às rotinas que implementam a classe; *protected*, que estende esse acesso a todas as suas

classes derivadas; *public*, que torna a manipulação da área ou utilização da operação totalmente livre. Uma operação associada a uma classe é chamada *função membro*.

A hierarquia de abstrações é construída em C++ através da definição de uma classe a partir de outra já existente. Nessa derivação é especificada a forma de herança das características da subclasse, ou seja, que tipo de visibilidade terão as componentes herdadas no novo tipo definido. Quando o programador usa na definição da classe uma função membro que poderá ser especializada em possíveis subclasses ele deve declarar esta função como sendo **virtual**; dessa forma o compilador se prepara para gerar código que trate durante a execução de associar a cada chamada dessa função a definição apropriada na hierarquia ao objeto a ser manipulado. Uma função virtual poderá ser redefinida em qualquer das classes descendentes, como abaixo:

```

classe A
{
    private:
    :           // Parte de A com acesso restrito às rotinas de
                // implementação. É o default para visibilidade
                // de membros da classe.

    protected:
    :           // Lista de membros que poderão ser acessados pelas
                // rotinas de A e de suas classes derivadas.

    public
        // Interface disponível para A
    virtual void f()
        { < código  $f_1$  > }
    :

};

classe B: public A    // B é subclasse de A.
                    // A parte pública de A continua pública para B.

{
    :
    public
    void f()
        { < código  $f_2$  > }
    :
};
  
```

```

classe C: public B
{
    :
    public
    void f()
        { < código  $f_3$  > }
    :
};

classe D : A
{
    :
    void g();
    :
}

```

As declarações acima definem que a operação f sobre objetos gerados pela classe A corresponde à execução da rotina f_1 , para objetos da classe B será executado o código f_2 e para a classe C a rotina correspondente à função virtual f é a f_3 . Observe que a classe D herda f de uma forma diferente (*private*) e portanto apenas as rotinas de implementação da classe, como g por exemplo, poderão utilizá-la.

Sobre herança múltipla, Stroustrup escreveu em 1983 que não considerava uma tarefa complicada permitir que uma classe fosse derivada de várias outras. Algumas pequenas modificações na forma de armazenamento e acesso a objetos deveriam ser feitas, já que acessar a parte referente a uma classe A em um objeto desta mesma classe ou em um objeto de uma classe derivada de A não poderia ser feito mais da mesma forma, como veremos a seguir. Em 1987 Stroustrup descreveu em [Str87b] um esquema para herança múltipla que julgava experimental, não considerando-o ainda como parte da linguagem C++. Atualmente a maioria dos compiladores C++ contam com herança múltipla, embora alguns acreditem que as vantagens decorrentes de sua utilização (como maior poder de expressão) não compensam eventuais complicações na manipulação das classes e objetos.

Implementação

A linguagem C++ tem o compromisso de ser mostrar compatível com a linguagem C, por isso todas as construções usuais deverão ser tratadas pelo compilador C++. Não há novidade em termos de modelo de máquina (o processamento continua baseado em acesso à variáveis e desvios que implementam as estruturas de controle); será possível

manter o esquema tradicional para endereçamento e chamada de rotinas?

Usualmente a geração de código para acesso a variáveis e chamada de funções é feita baseada em uma tabela de símbolos que reflete a estrutura de blocos da linguagem. Por exemplo, para que seja gerado código para a chamada $f(a)$ basta obter informação para endereçamento de a e f na tabela de símbolos e gerar as instruções necessárias para manipulação da pilha de execução.

Com a introdução de classes a geração de código deve levar em consideração a classe do objeto, já que muitas funções diferentes são denotadas pelo mesmo identificador e — mais importante — apenas funções definidas como parte da classe ou recebidas por herança poderão acessar os elementos da classe. Consideremos um exemplo com as seguintes classes definidas no programa:

Se temos as declarações

A a; B b; C c;

então os objetos a, b e c podem ter o seguinte aspecto de armazenamento:

Acumular a área herdada e estendê-la para compor o objeto de uma subclasse aproveita a idéia de *struct* em C e tem a vantagem de que o armazenamento do campo *i*, por exemplo, tem a mesma posição relativa ao início do objeto a ser manipulado, quer ele seja *a*, *b* ou *c*. Desta forma o endereçamento de um campo se mantém uniforme: a função *f* conseguirá acesso ao campo *i* sem precisar distinguir se tem em mãos um objeto da própria classe *A* ou de uma classe derivada desta.

Tratemos agora de um esquema responsável pela geração de código associado à chamada de funções membro:

```

a.f();           // Denota a operação f da classe A sobre o objeto a
b.g();           // idem, para g da classe B
c.h(3);          // idem, para h da classe C
b.f();           // Manipulação de objeto da classe B através de operação
                  // herdada de sua superclasse
c.f();           // Novamente utilização de operação herdada
b.M();           // Chamada não válida, pois M não faz parte das
                  // manipulações previstas para a classe B

```

Nos casos acima a determinação da rotina a ser chamada poderia ser feita após uma busca na definição da classe do objeto manipulado e na definição de seus ancestrais. Dessa forma, cada uma das operações acima poderia ser tratada como uma chamada tradicional de funções:

```

a.f() compilada como _A_f(a)
b.g()               como _B_g(b)
c.h(3)              como _C_h(c,3)
b.f()               como _A_f(b)

```

O problema se resume então a procurar a operação entre as explicitamente definidas e as herdadas e tratar o objeto manipulado como o primeiro parâmetro de cada função membro. Mas há casos em que esta tática não encontra corretamente a função associada a chamada, pois é possível redefinição de funções, ou seja, alguma função da classe é marcada com *virtual*. Focalizemos primeiro o caso em que não há funções redefinidas.

A tabela de símbolos, a fim de lidar com as situações que ilustramos através dos exemplos anteriores, deve associar a classe C às suas funções membro. Vemos três possibilidades:

- guardar todos os identificadores de funções membro numa tabela de símbolos onde um dos campos indica a qual classe pertence a função. Poderíamos então encontrar a operação desejada (sempre supondo que não há redefinição nas subclasses) através de uma busca nesta tabela. O grande inconveniente dessa solução é que o tamanho da tabela de símbolos para sistemas envolvendo muitas classes tornaria demasiadamente lenta a compilação;
- gerenciar uma tabela de símbolos para cada classe contendo a lista de todas as funções (próprias ou herdadas) que podem ser utilizadas envolvendo objetos de uma dada classe. Esta alternativa é muito ineficiente em espaço já que as funções de uma classe base fariam parte de todas as tabelas de suas subclasses;
- Manter uma tabela de símbolos contendo apenas as operações definidas *naquela* classe. A ligação com os membros herdados pode ser feita através de um apontador para a tabela de símbolos de sua superclasse imediata:

Esta última idéia pode ser associada à noção de escopo alocando uma nova tabela para cada novo nível de encaixamento da unidade; dessa forma a ativação de tabelas pode servir para toda a manipulação de símbolos, inclusive para variáveis globais e funções dissociadas de classes.

A discussão acima sugere que a utilização de uma função envolve a busca na interface da classe em questão e na de seus ancestrais. Para as classes que tenham função membro *virtual* esta busca já não é mais conveniente, como vemos no exemplo a seguir:

```
// Especificação da classe X
class X
{
    int i;
    public:
        virtual int f();
};

// Definição da classe Y, derivada de X
class Y: public X
{
    int i;
    public:
        int f(); // Especializa f redefinindo-a.
};
```

```
X x;           // criação de objeto x
Y y;           // criação de objeto y
X* px = &x;    // px aponta para x
Y* py = &y;    // py aponta para y
...
px = py;       // Como todo objeto de Y é um objeto de X um apontador
               // para X pode apontar para um objeto de Y
px -> f();      // Chamada da operação f
```

Para este exemplo $px \rightarrow f()$ deve ativar a chamada da função f especializada para a classe Y .

Observe que pelas informações disponíveis em tempo de compilação a procura de f se daria na tabela correspondente a X , já que a chamada envolve px , quando o esperado é que a manipulação do objeto apontado por px seja feita através da f definida para Y . Assim, percebemos que a geração de código para a chamada deverá levar em consideração o objeto dado como argumento em tempo de execução; dependendo do tipo do objeto a operação apropriada na hierarquia será chamada. Stroustrup sugere que para tratar deste caso seja inserida uma informação de tipo no objeto, mais especificamente um apontador para a tabela de símbolos particular a sua classe, que deverá conter *todas* as funções virtuais para objetos da classe. Essa tabela evitará a necessidade de se realizar em tempo de execução uma busca na hierarquia de tabelas. Objetos da classe X e Y terão então o seguinte aspecto:

Pensemos agora em como o esquema descrito acima se comporta para armazenamento e chamada de funções envolvendo **herança múltipla**:

`class C: A, B { ... }`

Um objeto do tipo C pode ser armazenado sob a seguinte forma:

Como antes, o compilador tem como gerar código para acesso a cada campo do objeto, pois conhece em tempo de compilação a distância deste ao início do objeto.

Já o esquema de chamada de funções não funcionará tão bem:

O comando $pc \rightarrow fb()$ causaria, pelo esquema sugerido anteriormente para herança simples, a geração de código para $_Bfb(pc)$. Mas observe que fb , sendo uma operação definida para B , utilizará o apontador recebido como base de um objeto B , o que não seria válido com o argumento pc . A chamada correta deveria ser

$$_Bfb(pc + \text{delta}_B)$$

onde delta_B é uma constante que indica a posição relativa de B em C . O compilador deve manter para cada classe uma constante delta junto com cada apontador para superclasse. Essa questão de endereçamento relativo aparece também na geração de código para *casting*:

B b;

C c;

B* pb = &a; resulta em

pb = (C*)pc deve resultar em

Assim, a operação de *casting* também envolve a constante de ajuste para base do objeto. Mais alguns detalhes devem ser tratados a fim de que esse esquema permita geração de código para herança múltipla de uma maneira aceitável em eficiência.

A questão do confronto de nomes entre as partes herdadas foi resolvida em C++ permitindo que o usuário utilize herança envolvendo conflitos, desde que na utilização dos nomes iguais seja resolvida a ambigüidade através do operador de escopo (::). Veja um exemplo a seguir.

C*pc;

⋮

pc ->(A :: f); // Ok

pc ->(B :: f); // Ok

pc ->(f); // Erro: ambigüidade

Referências

A linguagem C++ é descrita em [Str86a] e as idéias gerais podem ser encontradas de maneira resumida em [Str86b]. Vale a pena acompanhar o histórico da linguagem C++ através dos artigos [Str 82, Str 83] em que Stroustrup propunha mudanças para a linguagem C, pois percebemos quais construções da linguagem foram problemáticas e que linha foi seguida para chegar até as construções sintáticas que temos hoje. Ainda sobre o andamento do projeto temos um artigo sobre a evolução da linguagem no período de 1983 a 1985 [Str87c] e um que discorre sobre o futuro da linguagem [Str87d].

Em [Dew 87] há considerações, problemas e soluções sobre a implementação de tabelas de símbolos em compiladores C++ e tratamento de escopo. Descrição e comentários sobre aplicações desenvolvidas em C++ podem ser encontradas em [Wei 88], [Str 84], [Tri 88], [Gor 87], [Ca 86], [Wie 87], [Jer 89]. Em [Sak88a] encontramos uma coletânea de problemas detectados na linguagem.

No próximo capítulo usaremos a linguagem C++ como ferramenta para exploração do paradigma e discutiremos até que ponto seus mecanismos se adequam para expressar as principais idéias de programação por objetos.

4.1.2 Smalltalk

A história do sistema *Smalltalk* tem início na década de 70, quando pesquisadores da Xerox buscavam uma concepção nova para geração de programas. O objetivo era diminuir a distância entre o modelo que temos do mundo e o que introduzimos no computador, possivelmente através de uma nova forma de expressar as idéias presentes no processamento.

No próprio processo de desenvolvimento do sistema *Smalltalk* é possível destacar alguns pontos interessantes:

- O grupo de pesquisa almejava mais do que uma linguagem de programação de fácil utilização, havia uma preocupação geral com a questão de interface com o usuário. Existiam duas principais áreas de pesquisa: uma linguagem de descrição que servisse de interface entre as abstrações do programador e os modelos no hardware da máquina e uma linguagem de interação com o usuário que fosse compatível com a forma humana de comunicação. Certamente a linha de pesquisa era ambiciosa, mas se pensamos em como era a interação com a máquina há vinte anos atrás percebemos que eles queriam algo mais “amigável”, pelo menos fácil de usar.
- O trabalho de projeto do sistema *Smalltalk* se desenvolveu em ciclos, ao longo de quase quinze anos. Cada ciclo durava de dois a quatro anos e compreendia o

aperfeiçoamento do sistema corrente através da experiência com uma aplicação, conseqüentes reformulações e novas implementações. Smalltalk-80 foi resultado de cinco iterações desse processo.

- Os pesquisadores acreditaram na idéia de interfaces fortes, que restringissem as formas de acesso à estrutura de dados. Os princípios de tipos abstratos de dados já começavam a ser difundidos largamente, mas poucos acreditavam que eles se tornassem operacionais; nem mesmo nas primeiras versões da linguagem Simula 67 havia mecanismos para *information hiding*. Atualmente há quem julgue *interface* como sendo uma das peças chaves no sucesso da programação por objetos; em [CCHO89] é feita uma ligação entre interfaces, tipos e herança.

Uma idéia básica caracteriza o *Smalltalk*: **uniformidade**. O princípio básico que se aplica a qualquer parte do sistema é “*o mundo é formado de objetos que se comunicam através de mensagens*”. Qualquer componente do sistema *Smalltalk* ou qualquer valor presente na execução é um objeto: números, retângulos, diretórios de discos, compiladores, filas, pilhas ... Não há diferenciação entre uma função do sistema operacional e uma rotina projetada pelo usuário, ambas são simplesmente mensagens dirigidas aos objetos apropriados.

Um *objeto* em *Smalltalk* consiste de uma área na memória e um conjunto de operações, que são chamadas de *métodos*. Uma *mensagem* é uma requisição dirigida ao objeto para que este desenvolva uma de suas operações, da maneira que julgar adequada. A *interface* do objeto é definida pelo conjunto de mensagens a quais o objeto pode responder. Somente as operações na interface de um objeto podem manipular a área destinada ao mesmo.

Como vemos, as entidades básicas do sistema são os conceitos a que já nos dedicamos nos capítulos anteriores. A descrição de um objeto, isto é, especificação de armazenamento e conjunto de operações é feita através de *classes*. Todo objeto em *Smalltalk* é *instância* de uma classe.

Através de *subclasses* pode ser estabelecida a hierarquia entre as classes. O conceito de subclasse em *Smalltalk* baseou-se em construção similar na linguagem *Simula*. Uma subclasse especifica que suas instâncias são como instâncias da superclasse a menos de diferenças explicitamente especificadas na definição da superclasse. A princípio apenas herança simples pode ser expressa em *Smalltalk*, mas já há versões que trabalham com herança múltipla[I-B 82].

Considere o trecho seguinte da hierarquia de classes presente no sistema Smalltalk V (uma versão do Smalltalk-80 para microcomputadores):

```

Object
  Behavior
  ...
  Boolean
  ...
  Collection
    Bag
    Indexed Collection
      FileSizedCollection
        Array
        BitMap
        ByteArray
        CompiledMethod
        FileControlBlock
        Interval
        String
        Symbol
        OrderedCollection
          Process
          SortedCollection
    Set
      Dictionary
      ...
      SymbolSet

```

Observe que a classe Boolean, tipo usual em linguagens de programação, tem o mesmo status dentro do sistema que objetos do tipo FileControlBlock (estrutura usada pelo DOS para manipulação de arquivos). Em *Smalltalk* o usuário fica o tempo todo em contato direto com a hierarquia de classes, pois é percorrendo-a que novas classes são definidas, métodos são inseridos ou removidos.

Implementação

Muitos consideram o sistema *Smalltalk* inutilizável em virtude de sua ineficiência, consequência da implementação consistente de todo um sistema a partir de um único princípio. De qualquer forma, ao utilizarmos o sistema não percebemos onde terminam nossas ferramentas para abstração e começam as tarefas de execução da máquina.

O sistema *Smalltalk -80* é composto de duas componentes básicas:

- *imagem virtual*, composta de todos os objetos do sistema;
- *máquina virtual*, que consiste de dispositivos de hardware e rotinas em linguagem de máquina que manipulam os objetos da imagem virtual.

A tarefa do implementador é então criar uma máquina virtual que interaja com o seu ambiente básico de operação.

Os métodos (operações) especificados em *Smalltalk* são traduzidos por um compilador para uma sequência de instruções a serem passadas para o interpretador, que junto com a memória de objetos compõe a máquina virtual. Uma implementação para a máquina virtual pode ser encontrada em [Sm 80]¹. A seguir descrevemos algumas características da implementação do paradigma de objetos e mensagens no sistema *Smalltalk* :

1. alocação e desalocação de objetos

Os objetos são armazenados no *heap*, sendo alocada área de memória contígua para armazenamento durante a criação do objeto. Um objeto é destruído com a liberação de sua área. Para evitar que após uma sequência de criações e destruições de objetos a memória disponível fique fragmentada o sistema *compacta* periodicamente a memória, ou seja, move os objetos ainda em uso para o início do heap. Se cada referência ao objeto movido utilizasse diretamente o endereço deste no heap teríamos uma carga enorme de processamento para atualização de apontadores após uma compactação de memória. Para tratar essa questão é mantida uma tabela de objetos, ou seja, para cada objeto apenas uma referência à sua área de armazenamento no heap é armazenada.

Em *Smalltalk* um objeto é alocado explicitamente, através da mensagem *new*, mas nenhuma construção da linguagem causa desalocação de objetos. A não existência de mecanismos para especificação de destruição de objetos foi uma decisão tomada pela equipe de desenvolvimento da linguagem para evitar o problema da desalocação de um objeto a que outros ainda se referem. A fim de que a memória não seja consumida por objetos não mais em uso o sistema deve identificar os objetos inacessíveis e desalocá-los, ou seja, é necessária a utilização de técnicas de *garbage collection*. O livro de Goldberg e Robson [Sm 80] sugere (e fornece) duas implementações para o *garbage collection*:

- contagem de referências
É mantido um contador de referências para cada elemento da tabela de objetos; quando este contador atinge zero o objeto fica passível de destruição;
- marcação da área utilizada
Quando a quantidade de memória disponível atinge um nível crítico o sistema (mais especificamente a pilha do interpretador e a tabela de variáveis globais ao sistema) é percorrido exaustivamente e toda área referenciada é marcada; no fim do processo objetos que não foram marcados serão desalocados.

Segundo o trabalho de Duff [Duf 86], em implementações de *Smalltalk* com sistema de contagem de referências não otimizada chega-se a ter até setenta por cento do tempo total de execução dedicado a manutenção dos contadores.

¹A especificação é feita em *Smalltalk*

2. execução de rotinas associadas a mensagens

Quando uma mensagem é enviada a um objeto a interface da classe deste objeto é examinada em busca de um método cujo nome (seletor) se adeque à mensagem. Caso não se encontre nenhum método a busca passa a ser realizada na interface de sua superclasse, e assim sucessivamente. Dessa maneira, o *binding* da mensagem com a rotina a ser executada é feita apenas durante a execução. Esse atrelamento em tempo de execução do método à mensagem tem o inconveniente de que só durante a execução de um programa poderá ser constatada a passagem de operandos não esperados pelo método, ou seja, *Smalltalk* não trabalha com verificação de tipos.

A execução do método correspondente a uma mensagem envolve a criação de um objeto chamado *contexto*, cuja função é análoga a do registro de ativação [S-HM 83] manipulado na chamada de procedimentos. Mesmo após o término da execução do método o objeto *contexto* continua existindo (um depurador poderia utilizá-lo para obter informações), o que deixará sob encargo do sistema de *garbage collection* a liberação da área ocupada em consequência de cada mensagem enviada. Em [Duf 86] há a estimativa de que cerca de noventa e sete por cento da área alocada é destinada a *contextos*.

3. Para manter a uniformidade de componentes do *Smalltalk* uma classe também deve ser um objeto que atende a um conjunto de mensagens. Uma classe cujas instâncias sejam também classes é chamada de *metaclasses*. Toda vez que uma classe é definida uma nova metaclasses é criada para ela automaticamente² uma metaclasses. Metaclasses são semelhantes a outras classes porque contém os métodos usados por suas instâncias; diferem destas porque elas próprias não são instâncias de metaclasses.

Observe que a uniformidade ainda não foi totalmente atingida, pois como a criação de metaclasses é implícita há uma diferenciação entre o nível do usuário e do implementador. Em [B-C 89] é discutido o sistema *Classtalk*, que permite a criação explícita de metaclasses; são levantadas vantagens da utilização de metaclasses e discutidas as conseqüentes diferenças de enfoque na criação de classes a partir de metaclasses definidas pelo usuário.

A dissertação de doutoramento de Ungar ([Ung 86], [Ung 87]) descreve arquiteturas de hardware e técnicas de software que resultam em um sistema *Smalltalk* de alta performance. Trabalhando com um microprocessador SOAR conseguiram um desempenho muito superior a do *Smalltalk* -80 processando em um Motorola 68010, por exemplo. Muitos esforços têm sido realizados para tornar o sistema *Smalltalk* mais eficiente a fim de que este possa ser amplamente utilizado como “ambiente de programação exploratória”.

²Nas primeiras versões do *Smalltalk* o sistema era dotado de uma única metaclasses, chamada *Class*, responsável por toda manipulação envolvendo as classes (por exemplo, criação de objetos). Essa linha era muito restritiva pois eliminava a chance de atrelarmos a uma classe rotinas de inicialização especiais para aquela particular classe.

Ambiente

Smalltalk possui uma enorme variedade de classes e métodos definidos e a cada programa desenvolvido pelo usuário o sistema se torna maior. Para que os recursos disponíveis se tornem utilizáveis são imprescindíveis ferramentas para obtenção de informações acerca dos objetos definidos no sistema. O ambiente *Smalltalk*, todo baseado em janelas, conta com:

- *inspect window*: Ao receber a mensagem *inspect* um objeto terá sua estrutura (definição de classes, métodos, subclasses, superclasses) mostrada em uma janela. É uma ferramenta importante para depuração de programas (pois permite observar a origem e forma de um objeto qualquer) e para manter o caráter exploratório da linguagem, pois olhando para a definição de um objeto do sistema podemos obter mais informações sobre alocação, inicialização, indexamento, etc.
- *browse*: o sistema *Smalltalk* conta com uma classe chamada *Browse*, cujos objetos funcionam como ferramentas para pesquisa na “enciclopédia de classes” definidas. Sem um tal mecanismo conveniente para recuperação de informação o usuário se perde no emaranhado de métodos e subclasses. Para criar uma definição nova é necessário acessar a hierarquia no local desejado para a nova classe, introduzir a nova definição na árvore e passar a definir as instâncias e métodos. No momento que fornecemos a descrição da rotina associada a uma mensagem a mesma é compilada e armazenada junto com as demais definições existentes.
- *notify window*: o sistema conta com objetos para auxiliar na depuração de programas; o usuário pode especificar pontos de parada de execução e o sistema então providenciará uma janela para investigação de valores. Nossa experiência considerou a atividade de depuração em *Smalltalk* muito cansativa pela grande quantidade de janelas abertas para notificação que deverão ser fechadas pelo usuário.

Trabalhando com Smalltalk

Exemplos interessantes de classes e métodos em *Smalltalk* podem ser encontrados no excelente livro de Goldberg e Robson [Sm 80]. O livro de Kaehler e Patterson [K-P 86] tenta introduzir a filosofia de programação em *Smalltalk* a partir de problemas clássicos em programação. Em [N-R 89] dois pesquisadores contam sua experiência em penetrar no mundo *Smalltalk* e apontam alguns pontos que consideraram prejudiciais para o entendimento e uso do sistema.

4.1.3 Outras Linguagens

Há uma grande variedade de linguagens que por uma característica ou outra se relacionam com o paradigma dos objetos. Algumas são realmente interessantes, como *Eiffel* e *Trellis-Owl*. Muitas surgiram de uma mistura (a primeira vista improvável) de paradigmas de computação, onde a idéia principal é estender a noção de derivação para os mecanismos de abstração do paradigma. Enumeramos a seguir as que consideramos de maior destaque na literatura corrente.

Eiffel

A linguagem *Eiffel* ([Mey87c],[Mey 88]) foi desenvolvida sob a égide da reutilização de software, concebendo um sistema como sendo uma coleção estruturada de implementações de tipos abstratos de dados. A definição de um tipo abstrato em Eiffel é feita com a declaração `class`, que envolve a definição da interface do tipo e uma definição de estrutura de dados e rotinas. Há dois pontos interessantes na linguagem:

- herança múltipla[Mey87b]
Uma classe pode ser criada a partir de várias outras não correlatas, isto é, possuir várias superclasses imediatas na hierarquia. O problema de conflito de nomes deve ser solucionado pelo usuário, que fornecerá novos identificadores para as entidades com nomes idênticos nas superclasses.
- *invariantes*, *pré* e *pós* condições
Através das palavras-chave `require` e `ensure` podem ser especificadas condições a serem verificadas na entrada e saída de rotinas; com `invariant` representa-se restrições de consistência que serão impostas a todas as rotinas.
- classes genéricas
Estão disponíveis três classes na biblioteca para lidar com coleções de elementos de qualquer tipo:

```
ARRAY[T]  
LIST[T]  
LINKED-LIST[T]
```

T é um parâmetro formal representando um tipo arbitrário; dessa forma poderemos ter listas de tipos quaisquer.

Não há duplicação de código na implementação desses tipos polimórficos, ou seja, é gerada apenas *uma* rotina de inserção, *uma* para remoção, etc tanto para `LINKED-LIST[int]` quanto para `LINKED-LIST[float]`.

A implementação de *Eiffel* é feita através de uma tradução do programa fonte para a linguagem C e, segundo Meyer, o tempo gasto com a tradução leva aproximadamente o mesmo tempo que o compilador C levará para gerar o código objeto. Como a

tradução de *Eiffel* para C é feita classe por classe (gerando diferentes arquivos e portanto código objeto separado por classe) o reaproveitamento de classes não aumenta a carga de trabalho do tradutor.

O caso de redefinição de funções herdadas é tratado de maneira eficiente, já que a busca (em tempo de execução) da definição apropriada no grafo de hierarquia (como *Eiffel* suporta herança múltipla a relação entre as classes tem estrutura mais complexa do que uma árvore) é feita em tempo constante, independente do diâmetro do grafo e utilizando pouco espaço extra (embora os artigos sobre a linguagem não entrem em detalhe sobre como é feito o *binding* da função correspondente ao objeto, provavelmente a estrutura do objeto é expandida de maneira a guardar alguma informação acerca do tipo que será consultada em tempo de execução e é mantida uma tabela de operações disponíveis em cada classe da hierarquia). Para gerenciamento de memória *Eiffel* conta com um sistema de alocação e desalocação de objetos e com um processo (concorrente) para *garbage collection*.

O ambiente para *Eiffel* inclui dois utilitários interessantes: *short*, que produz uma versão resumida da descrição de cada classe, e *ep*, que faz modificações opcionais no código C gerado a fim de obter maior eficiência.

Sendo programas em *Eiffel* traduzidos para C é bem provável que a depuração de tais programas fique bem prejudicada.

Trellis-Owl

A linguagem *Trellis-Owl* [Sch86] é a base de um ambiente completo para programação por objetos: *Trellis*. Foi projetada de maneira a facilitar o projeto, implementação e evolução de programas grandes. A noção de tipos e hierarquia de tipos proporciona uma base compreensível para futuras equipes que darão manutenção ao sistemas. *Trellis-Owl* combina uma hierarquia de tipos que engloba herança múltipla com verificação de tipos em tempo de compilação. A linguagem utiliza uma sintaxe convencional e emprega a notação usual de chamada de procedimentos para manipulação de objetos. A biblioteca pré-definida da linguagem define os tipos básicos como inteiros, *strings* e vetores. As operações em *Trellis-Owl* são de duas espécies: sobre objetos e sobre classes; é através de operações sobre classes que objetos são criados. É permitido a sobrecarga³ de operadores.

Subtipos em *Trellis-Owl* foram concebidos para expressar especialização de comportamento, “não uma hierarquia de implementação como em Smalltalk”. Através de herança é possível reutilização de comportamento com a possibilidade de redefinição de rotinas herdadas. No caso de herança múltipla o conflito de nomes deve ser solucionado pelo usuário, quer seja escolhendo a origem da rotina entre as várias superclasses ou redefinindo a operação em questão. Outros pontos que merecem destaque são:

³ *overloading*

- construtor para tipos

Em *Trellis-Owl* podemos fazer:

```
x: Integer | String;
```

```
print(x);
```

O objeto *x* poderá armazenar tanto um inteiro quanto um real, e a escolha da função *print* apropriada se faz em tempo de execução.

- geradores de tipos

Um gerador de tipos é uma única definição para uma família de tipos quase idênticos; é a descrição parametrizada de um tipo, por exemplo:

```
type_module Set [Element_Type : Type]
```

```
operation create (mytype)
```

```
  returns (mytype);
```

```
  ! Cria um conjunto.
```

```
operation insert (me, element: Element_Type)
```

```
  ...
```

```
  ! Insere um elemento no conjunto
```

```
operation extract (me)
```

```
  returns (Element_Type);
```

```
  ! Devolve um elemento arbitrário do conjunto
```

```
end type_module;
```

O gerador de tipos *Set* define uma variedade de tipos: *Set[Integer]*, *Set[String]*, *Set[Set[real]]*.

- iteradores

Iteradores permitem extrair o conteúdo de uma coleção item por item. De uma maneira geral iteradores fornecem uma série de objetos, sem revelar a forma de implementação desta coleção. A idéia está presente por exemplo em C++ , implementada através de uma classe que guarda o estado no percurso da coleção e operações que iniciam o percurso, avançam, devolvem o atual elemento e testam fim de percurso. Mas em *Trellis-Owl* aparece de forma mais limpa, ligado a um comando de iteração:

```
for c: Character in chars ("oi mundo") do
```

```
  print (c);
```

```
end for;
```

A operação *chars* deve ser definida pelo usuário, por exemplo:

```
operation chars (me)
```

```
  yields (Character)
```

```
  is
```



```

begin
  var i: Integer := 1;
  loop
    exit when i > me.size;
  end loop;
end;

```

Observe que os dois loops (o *for* que imprime e o *loop* em *chars*) interagem, funcionando como co-rotinas.

- tratamento de exceções

Existem formas consistentes para especificação de rotinas que atendem a tratamento genérico de erros e condições especiais que se apliquem ao sistema como um todo.

- “type cases”

É possível obter execução condicional onde o teste é feito no tipo da variável em questão:

```

...
type_case X
  on Integer do
    ...
  on Real do
    ...
  otherwise do
    ...
end type_case;

```

Esta parece ser uma construção interessante da linguagem, mas se a hierarquia de tipos estiver bem construída o próprio mecanismo de herança e redefinição de operações obterá o efeito de escolha conveniente para código com base no tipo do elemento envolvido. Essa construção é um caso tipo de polimorfismo *ad-hoc* (seção 3.3). Não sabemos como essa construção se comporta quando testamos o tipo de um objeto em meio a uma hierarquia, ou seja, se o teste engloba o aspecto de especialização de tipos respondendo verdadeiro para todos os testes envolvendo ancestrais, mas acreditamos que sim. Nossa impressão após considerar possibilidades de simulação para tal construção em C++ ou *Smalltalk* é que uma carga enorme de processamento estaria envolvida na manutenção de interfaces que envolvessem esse tipo de testes se o compilador não armazenasse o grafo correspondente à hierarquia para consultas durante a execução.

Segundo a equipe que há mais de quatro anos trabalha com a linguagem, *Trellis-Owl* consegue aliar bom poder de abstração e eficiência na execução.

Beta

Beta [Beta] foi projetada para ser uma linguagem de programação moderna na linha de SIMULA. *Beta* usa um único mecanismo de abstração, chamado *padrão*⁴ para denotar classes, procedimentos, funções e tipos. Os padrões funcionam como modelos para geração de objetos, pois especificam valores utilizados para armazenamento, parâmetros de entrada e saída e seqüência de comandos associados ao padrão. É possível especificar uma hierarquia de padrões, através do mecanismo de *subpadrões*.

A idéia de englobar no conceito de padrão tanto atributos quanto especificação de funções torna o mecanismo de herança mais poderoso. Ao criar um subpadrão para um padrão correspondente a uma função, por exemplo, estaremos especializando a especificação da função [KMPN87]. Analogamente é possível especializar um atributo qualquer de um padrão e assim conseguir classes parametrizadas [M-P 89].

Beta tem mecanismos para *alternação* (uma seqüência de co-rotinas) e concorrência [KMPN89]. *Beta* deverá fornecer no futuro suporte para programação funcional e programação em lógica.

Uma versão experimental de *Beta* foi implementada em várias máquinas da linha Motorola 68000. Os responsáveis pelo desenvolvimento de *Beta* esperam que *Beta* seja utilizada para desenvolvimento de sistemas grandes, inclusive sistemas distribuídos.

Oberon

Segundo Wirth em [Wil 88] a linguagem *Oberon* surgiu durante o projeto de um sistema operacional para estação de trabalho mono-usuário que fosse moderno, flexível e eficiente. A idéia original era usar a linguagem *Modula*, que fornece uma boa base para desenvolvimento de sistemas com módulos de interfaces bem definidas. *Modula*, porém, não incorpora *garbage collection* nem possui uma maneira adequada para extensão de tipos já definidos pelo usuário, pontos que se revelaram importantes para o desenvolvimento do sistema dentro dos princípios traçados.

Oberon resulta da transformação de *Modula* através do acréscimo de algumas construções e da eliminação de outras que tornavam a linguagem e a compilação mais complexas sem aumentar de maneira significativa o poder de expressão ou abstração da linguagem. Um artigo de Wirth ([Wil 88]) discute em detalhes os objetivos e a linha de desenvolvimento da linguagem.

Em [Wi2 88] há a definição da linguagem. Não estão presentes os usuais conceitos de classes, objetos e herança. A idéia de encapsulamento e módulos estão presentes, mas não na forma usual para programação por objetos já que não há um atrelamento sintático da estrutura de dados e as operações que as manipulam. A idéia de herança

⁴*pattern*

propriamente dita também não está presente, embora exista uma hierarquia de tipos (tipos podem ser criados como extensão de outros; se T_2 é extensão de T_1 então as expressões declaradas como tendo o tipo T_2 podem ser utilizadas como valores associados a T_1). Está definido o operador *is* que verifica se uma dada variável é de um determinado tipo; o operador pode ser utilizado para a implementação de rotinas que manipulem alguns tipos distintos.

Baseados na nossa concepção do paradigma não consideramos *Oberon* uma linguagem orientada a objetos, já que não é possível especificar especialização de comportamento.

Clu

Clu [Lis 77] foi projetada para programação com tipos abstratos de dados, fornecendo mecanismos para descrição de tipos e interfaces de maneira que as abstrações fossem encapsuladas em módulos. As idéias desenvolvidas durante o projeto e implementação serviram de base para o desenvolvimento de linguagens realmente orientadas a objetos.

O modelo de computação de *Clu* é composto de *objetos* e *variáveis*; objetos são os valores que aparecem no programa e variáveis são os nomes utilizados no programa para referenciá-los. Variáveis distintas podem compartilhar objetos e dessa forma englobar visões distintas do mesmo elemento.

Os objetos em *Clu* são automaticamente criados (na primeira referência ao valor) e teoricamente continuam a existir durante toda a execução; na prática *Clu* tem um sistema que detecta os objetos que não estão mais sendo referenciados e libera a área associada a eles. Alguns objetos são tidos como mutáveis, pois podem ter características de comportamento alteradas. É o caso dos objetos com área de armazenamento associada e operações que alteram seu valor. Objetos que não apresentam comportamento variável com o tempo (como por exemplo *integer*, *boolean*, *strings*) são chamados de *tipos constantes ou imutáveis*. A noção de tipo mutável e imutável é interessante, já que separa bem os casos em que as operações são realmente funções, ou seja, associam valores a seus operandos sem capacidade de transformar o valor fornecido.

Objetos em *Clu* são implementados através de um descritor de tipos, que é um valor de tamanho fixo que armazena um código para um tipo e as informações que descrevem o tipo. Quando o tipo é mutável são armazenados apontadores para a área utilizada para os valores. O código do tipo é usado pela rotina de *garbage collection* em sua busca de elementos disponíveis para liberação de área e pelo depurador. Como *Clu* não trabalha com herança este código não é necessário para determinação da rotina apropriada a uma dada operação da interface.

Emerald

Emerald [Hut 87] é uma linguagem baseada no modelo Smalltalk, onde todas as entidades são objetos. Um objeto em *Emerald* carrega uma identidade dentro da rede, de forma que atualizações nas várias visões do objeto são mantidas consistentes; há encapsulamento da representação do objeto. Um objeto pode opcionalmente conter uma seção de inicialização que especifica um conjunto de operações a serem executadas durante a criação do objeto e uma seção para um processo que pode ser criado durante a inicialização.

Foi projetada especificamente para sistemas distribuídos.

Objective-C

Objective-C [Cox 84] é uma extensão da linguagem C que procura expressar os conceitos de Smalltalk sem utilizar seu modelo de máquina. Ao contrário de C++ enfatiza o polimorfismo e o *binding* dinâmico. Sua biblioteca de classes usou o ambiente Smalltalk como modelo. A linguagem está disponível para muitas máquinas, sendo utilizada para o desenvolvimento de muitos sistemas orientados a objetos.

Baseadas em Lisp

Várias extensões para a linguagem Lisp envolvendo classes e herança foram projetadas: CommonObjects[Sny 86], Flavors[Moo 86], Ceyx[Hul 84], CommonLoops [Bob 86], ObjectLisp, e Common Lisp Object[Bob 88].

Wolf em [Wolf89] compara a linguagem Flavors com C++ através de projetos para descrever e gerar hardware eletrônico. O trabalho é muito interessante, principalmente para os que investigam vantagens ou desvantagens de trabalhar com linguagens com sistema de tipos rígido.

Baseadas em Programação em Lógica

Algumas linguagens usam Prolog como base, com o objetivo de obter uma linguagem de programação em lógica que generalize a idéia de unificação de maneira a levar em conta o reticulado que representa a relação entre tipos. Os objetos passam a englobar regras, fatos e consultas. Um exemplo é a linguagem LOGIN[A-K 86]. Há projetos como *Alf* [Mel 89] que fornecem definições de classe para servir de integração entre Smalltalk e Prolog.

Ada

É possível encontrar dezenas de trabalhos sobre programação por objetos em Ada ([Mey 88], [Bo 86], [Lad 88]). Como as idéias de tipos abstratos e encapsulamento estão presentes em Ada de uma maneira bem completa, muitos pesquisadores consideraram conveniente emular as técnicas de programação por objetos nesta linguagem.

Pascal

Em [J-K 87] encontramos uma aplicação desenvolvida sob a filosofia de objetos e interface estrita. Segundo os autores a utilização de linguagens especializadas pode ser substituída por uma disciplina rígida na utilização de valores (em geral *records*, considerados como os objetos do sistema). O sistema descrito no trabalho tratava de simulação gráfica de tratamentos para câncer através de radioterapia.

Há duas extensões de Pascal que são voltadas para objetos: Clascal e Object Pascal [Tes 85].

Recentemente compiladores Pascal muito populares (Turbo da Borland, MicroSoft Pascal) passaram a englobar classes e operações virtuais. Na linha de abstração obtida através de disciplina na utilização de rotinas encontramos também um artigo sobre tipos abstratos em Fortran [Isn 82] e em Forth [Pou 86].

4.2 Algoritmos para herança múltipla

Durante algum tempo as equipes que desenvolviam linguagem voltadas para objetos optaram por não englobar herança múltipla como mecanismo para reutilização de unidades. Certamente a principal restrição era quanto à implementação: a estrutura geral de *grafo* ao invés de árvore hierárquica para abstrações parecia comprometer seriamente a busca de informação acerca das entidades herdadas, tornando o processamento ineficiente em termos de tempo e espaço.

Todas as linguagens que oferecem construções para subclasses têm sua estrutura de geração de código fortemente baseada em duas premissas: a hierarquia é conhecida em tempo de compilação (ou execução, para linguagens interpretadas) e obter a seqüência de ancestrais para uma dada classe é operação corriqueira. De fato, se trabalhamos apenas com herança simples toda a informação sobre derivação de classes pode ser armazenada em uma árvore e a seqüência de superclasses para uma dada classe é dada pelo caminho desta até a raiz. Quando estendemos o poder de reutilização de forma a englobar mais de uma superclasse imediata na hierarquia a informação deverá ser obtida de um grafo dirigido (acíclico) e temos então uma nova questão: que forma é apropriada para percorrermos o grafo procurando, por exemplo, funções eventualmente herdadas.

As duas maneiras usuais para percurso em grafos — busca em largura e busca em profundidade — são utilizadas por algumas linguagens, mas sob alguns aspectos não são convenientes já que não há garantia de que a superclasse raiz da hierarquia seja a última classe visitada. Para o grafo seguinte, por exemplo, as duas formas de busca visitariam o vértice correspondente a classe *A* no meio do percurso.

Uma ordenação conveniente para tais hierarquias pode ser obtida através de um algoritmo de ordenação topológica (cuja complexidade é $O(VG + AG)$, onde VG é o número de vértices do grafo representando a hierarquia e AG seu número de arestas). O trabalho de Ducournau e Habib [D-H 87] enfoca em detalhe a questão de algoritmos para obtenção da ordem de percurso, apresentando dois algoritmos que também refletem a idéia de ordenação topológica.

Uma forma possível de evitar percursos em grafos seria armazenar para cada classe uma lista completa de operações disponíveis; dessa forma as informações estariam centralizadas em cada ponto do grafo. A construção desta lista seria simples, pois bastaria concatenar listas de superclasses (onde estariam especificados os identificadores e endereços das rotinas herdadas) e atualizar o endereço de funções redefinidas. Já a utilização da “central de informações” da classe poderia consumir tempo e espaço em demasia. Além da repetição de informação pelos nós do grafo, como já discutimos na seção 4.1.1, esta forma de armazenamento pode consumir muito tempo na recuperação de informação se a estrutura de dados não oferecer uma forma eficiente para acesso. Uma solução para a obtenção em tempo constante do endereço correspondente ao identificador de uma função é associar a cada identificador um número e construir para cada classe um vetor característico para o seu conjunto de operações. Dessa forma, pode-se em tempo constante verificar se uma dada função faz parte da interface e ainda obter o endereço da função em caso afirmativo. Acompanhemos essa estrutura de armazenamento através do seguinte exemplo:

Uma tabela que associa um número a cada uma dos identificadores de funções presentes no sistema poderia ser dada por:

identificador	número associado
a	1
b	2
c	3
d	4
e	5
f	6
h	7

De uma maneira geral o espaço ocupado nesta forma de armazenamento para os endereços das funções – supondo que um endereço ocupe uma palavra – seria
número-de-classes * *número-de-seletores-de-operações* palavras

Se pensamos em um sistema que utiliza 100 classes e 500 operações teríamos 50000 palavras dedicadas ao trabalho do compilador (que podem continuar presentes na execução a fim de tratar das funções virtuais), o que em alguns casos inviabilizaria a utilização do sistema.

Em [DMSV89] encontramos uma maneira interessante de diminuir o tamanho do vetor característico de cada classe, que pelo esquema acima possui *número-de-seletores* posições mas que utiliza apenas o número correspondente a quantidade de operações de que dispõe. Sua solução se baseia na seguinte idéia: se os seletores *x* e *y* nunca aparecem simultaneamente em uma classe então podem ocupar a mesma posição no vetor característico, ou seja, um único número seria associado a ambos. Basta então numerar os seletores dentro do menor intervalo possível, de forma que seletores da

mesma classe tenham sempre números distintos.

A idéia acima remete a um problema clássico em teoria dos grafos: coloração de vértices. Construimos um grafo onde cada vértice representa um seletor e vértices são adjacentes se e somente se existe uma classe com os dois seletores em sua interface; bastará então obter uma coloração dos vértices do grafo para que tenhamos uma numeração conveniente para os seletores. De preferência a coloração deve utilizar o menor número possível de cores.

Para o nosso exemplo o grafo de seletores associado é:

Uma possível coloração (numeração de seletores) deste grafo é dada por:

vértice (<i>seletor</i>)	cor (<i>número associado</i>)
a	1
b	2
c	3
d	4
e	2
f	1
g	1
h	1

Cada classe passará a guardar quatro posições para endereços, ao invés de sete. Observe que a coloração acima usou o menor número possível de cores. A determinação de uma coloração com número mínimo de cores é um problema computacionalmente difícil (*NP-completo*), mas existem heurísticas que para a maioria dos grafos conseguem chegar bem perto do número mínimo.

Outros algoritmos para compactação de tabelas esparsas poderiam ser utilizados a fim de diminuir o espaço ocupado para representação da interface de uma classe. Para ambientes onde memória é um fator crítico talvez a melhor solução seja sacrificar eficiência no acesso armazenando cada lista de operações no menor espaço possível, ou seja, para cada classe a área de armazenamento de endereços de rotinas e o tempo de acesso seriam proporcionais ao número de operações disponíveis na interface.

Capítulo 5

Uma Experiência com Programação por Objetos

Até aqui nosso trabalho investigou o paradigma de programação por objetos focalizando sua base de formação, ou seja, seus objetivos, conceitos e linguagens. A julgar pela quantidade de trabalhos publicados recentemente o paradigma ainda se apresenta como um forte campo de pesquisa para as áreas de semântica e implementação de linguagens de programação e ambientes para desenvolvimento de software.

Muitos indícios nos levam a acreditar que as idéias de classes e herança extrapolaram os grupos de pesquisa e desenvolvimento onde nasceram para hoje fazerem parte do panorama geral de produção de software. Grande parte dos anúncios divulgando novos utilitários e sistemas enfatiza de alguma forma a presença de *objetos*, como se eles garantissem software de boa qualidade utilizável em um espectro geral de aplicações.

Programar por objetos *está na moda* e parece ter tudo para atingir uma posição de destaque dentro do atual quadro de desenvolvimento de programas, nem que seja para servir de paliativo contra a sensação de impotência diante dos problemas da produção de software. Uma vez seduzidos pelo mundo dos objetos, o que faremos para efetivamente conquistá-lo? Por onde se começa a adotar um novo paradigma de programação?

A nossa experiência com outros paradigmas não nos serviu de modelo para essa nova iniciação. Quando trabalhamos com programação funcional tínhamos em mãos uma aplicação para a qual as *S-expressões* do Lisp e a idéia de processamento como mapeamento entre domínios se encaixavam perfeitamente. Além disso a própria linguagem exigia uma integração com o paradigma, já que não dispúnhamos das estruturas usuais para execução condicional e repetições. Também quando desenvolvemos um pequeno sistema em Prolog não nos vimos aprendendo uma nova forma de pensar em programas de uma maneira geral, apenas um novo enfoque para uma aplicação

fortemente baseada em regras.

O contato com as idéias de programação por objetos — através da leitura de artigos e livros — não foi suficiente para que pudéssemos discernir quais mudanças sua utilização traria ao processo de desenvolvimento de programas, mas já acreditávamos que a experiência não deveria se dar amarrada a um sistema ou linguagem em particular. Dessa forma descartamos a opção de começar o aprendizado utilizando *Smalltalk* antes mesmo de dispormos deste ou qualquer outro sistema com objetos, pois a linguagem nos levaria a empregar seu molde de desenvolvimento de programas antes que absorvêssemos as idéias de tipos abstratos de dados e encapsulamento como princípios gerais para obtenção de software reutilizável.

Neste capítulo descrevemos como se deu nosso contato prático com a programação envolvendo objetos, classes e herança. Relatamos o caminho percorrido desde a resistência inicial ao paradigma até a fase em que este se tornou natural (pelo menos para o tipo de aplicação a que nos dedicamos). Não há a sensação de nenhuma grande descoberta ou novidade, mas a mudança de perspectiva nos permite enxergar uma estruturação do sistema e do próprio processo de desenvolvimento dos programas que pode ser utilizada para obtenção de uma nova metodologia de desenvolvimento.

Durante o projeto lidamos mais de perto com a aplicação e implementação dos principais conceitos, podendo portanto perceber que em alguns casos não encontramos nas idéias de programação por objetos uma forma conveniente para expressar a relação desejada entre as unidades do sistema. Outras vezes constatamos que a linguagem utilizada não fornecia subsídios para a nossa modelagem, o que nos permitiu tirar algumas conclusões sobre seus mecanismos ou isolar dificuldades usuais em modelar o sistema de maneira conveniente para reutilização de especificação através de herança.

A escolha de nosso software “cobaia” se mostrou particularmente feliz, pois permitiu que tratássemos da questão de múltiplas representações (estruturas de dados) simultâneas para um mesmo objeto.

A linguagem C++ foi utilizada para implementação do sistema.

5.1 O projeto: uma biblioteca para grafos

Sob alguns aspectos o paradigma da programação por objetos é apresentado com uma *solução*, mas não considerávamos clara a espécie de problema que seu uso atacaria. Queríamos então, através da utilização da suas idéias em um programa, entender melhor o que seriam os conceitos abordados e ainda conseguir diretrizes para obtenção de respostas às seguintes perguntas:

- As classes já existem, saltando aos olhos na análise do domínio de aplicação, ou são fabricadas para servirem de modelo ao software?

- Até que ponto os dados se prestam a especialização? Novamente, será fácil detectar hierarquia na modelagem?
- De que maneira pode ser feita a extensão de um software desenvolvido com classes e herança? É possível que a criação de novas unidades seja feita conhecendo-se apenas a interface das classes já definidas?
- É fácil conseguir polimorfismo?
- Quanto seria simplificada a implementação de novas abstrações se já tivéssemos disponíveis classes para manipulação das estruturas de dados usuais (lista ligada, árvores, etc)?
- Qual o nível de independência entre as unidades do sistema? Até que ponto podemos mexer na implementação de uma unidade sem alterar sua interface?

Para o tipo de “validação” que queríamos fazer achamos que o mais conveniente seria o desenvolvimento de uma biblioteca, pois este tipo de software parece se adequar mais facilmente a idéia de valores e funções associadas, permitindo assim que seja explorada a noção de hierarquia. Além disso, no projeto de uma biblioteca seria mais fácil abstrairmos para cada operação um contexto geral de uso obtendo unidades reutilizáveis.

A escolha recaiu sobre uma biblioteca para manipulação de grafos. Tal tipo de ferramenta para desenvolvimento de programas que utilizem grafos já havia sido discutida algumas vezes com grupos de iniciação científica e se mostrava à primeira vista uma aplicação adequada para envolver tipos abstratos de dados, encapsulamento e criação de novas abstrações a partir de classes de valores já modelados.

5.1.1 Motivação

Os conceitos de teorias dos grafos podem ser utilizados para modelar alguns problemas de modo que a obtenção de uma solução se reduza à execução de um algoritmo canônico para manipulação de grafos. Muitas vezes o grafo obtido com a redução e o algoritmo empregado possuem características que, se bem exploradas, garantem uma solução eficiente para o problema original. Nesse tipo de situação a forma escolhida para representação dos grafos utilizados no programa exerce um papel fundamental no processamento. Em alguns casos a programação com grafos é menos compromissada com a eficiência, mas o trabalho envolvendo armazenamento ou manipulação destes ainda é responsável por grande parte da atividade de desenvolvimento: será necessário definir uma estrutura de armazenamento para os grafos, codificar o algoritmo em uma seqüência de chamadas que operem sobre a estrutura de dados e preparar uma forma de entrada de dados (em geral programando um gerador de instâncias com as características desejadas). Como resultado obtemos um programa onde a visibilidade das idéias importantes do algoritmo fica prejudicada pela presença de apontadores, índices e endereços.

Resolvemos explorar o mundo dos objetos através do desenvolvimento de uma biblioteca para manipulação de grafos onde estejam disponíveis as formas usuais para sua representação e manipulação. O software se propõe a evitar que o usuário se preocupe com estrutura de dados; para isso será definido um armazenamento *default* e ainda construções que permitam ao usuário escolher a estrutura de dados que julgue apropriada, sem que isso implique em alterações na forma de utilização pelo usuário das operações disponíveis. O objetivo é fornecer subsídios para que a forma de especificação do programa se torne mais parecida com o algoritmo tratado, deixando para a implementação da biblioteca o tratamento das questões de representação do grafo e codificação das operações. Uma vantagem da especificação do algoritmo sem vinculação com forma de representação é que várias alternativas para armazenamento da entidade poderiam ser testadas sem necessidade de alteração no programa.

Um trabalho na mesma linha foi desenvolvido em 1969 pelo Departamento de Ciência da Computação da Universidade de Houston. O software desenvolvido foi chamado de Graspe [FDFP69] e compreendia um pequeno conjunto de primitivas em Lisp.

5.1.2 Primeiro contato: análise informal de requisitos do software

A fim de dar início ao desenvolvimento de uma biblioteca de tipos abstratos devemos fixar quais são seus objetivos e que tipo de funcionalidade queremos oferecer a seus usuários. No nosso projeto assumimos como usuários os pesquisadores em teoria dos grafos e fizemos um levantamento dos requisitos do sistema a partir de descrições para algoritmos encontradas em vários livros.

De uma maneira geral, algoritmos em grafos envolvem *conjuntos*, *seqüências* e *grafos*. Para cada uma dessas entidades é possível de imediato vislumbrar algumas manipulações que deverão ser tratadas pelo software:

1. conjuntos

As manipulações canônicas devem estar disponíveis, como por exemplo:

- união, intersecção, diferença e diferença simétrica entre dois conjuntos;
- teste de igualdade e inclusão entre conjuntos;
- inserção e remoção de elementos;
- teste de pertinência;
- obtenção da cardinalidade.

As operações enumeradas acima aparecem após uma análise superficial da estrutura abstrata para conjuntos. Se pensarmos em conjuntos com características

próprias outras operações podem surgir, por exemplo para conjuntos de conjuntos podemos querer testar a se um dado conjunto é minimal/maximal ou obter a intersecção ou união dessa família de conjuntos.

Para os conjuntos cujos elementos apresentem uma relação de ordem total devem estar presentes as operações de *máximo* e *mínimo*. Deve ser possível também trabalhar com conjuntos homogêneos, ou seja, o sistema garantiria a consistência permitindo apenas inserção de elementos de uma dada espécie; dessa forma os algoritmos poderiam ter definidos conjuntos de vértices, conjuntos de caminhos em um grafo, conjuntos de grafos, etc. Deve também ser fornecido um mecanismo para percorrer um conjunto.

2. seqüências

As operações usuais sobre seqüência são:

- inserção e remoção de elementos na seqüência;
- obtenção do *i-ésimo* elemento da seqüência;
- obtenção de um segmento da seqüência;
- concatenação de seqüências.

Algumas seqüências especiais devem ser manipuladas: passeios, trilhas, caminhos e circuitos de um grafo.

3. grafos

Observando que tipo de manipulação em geral é feita nos grafos percebemos que não será suficiente fornecer ao usuário uma lista *razoável* de operações; o importante é definir um conjunto de operações básicas — expressas de uma forma parecida com o próprio linguajar do assunto — a partir das quais seja possível construir novas funções para manipulações específicas. Devem estar disponíveis:

- operações para inserção e remoção de vértices;
- operações para inserção ou remoção de arestas;
- operação para obtenção dos extremos de uma aresta (início e fim, no caso de grafo orientado);
- obtenção do conjunto de vértices adjacentes a um vértice;
- obtenção do conjunto de arestas que incidem em um conjunto de vértices;
- obtenção do conjunto de vértices cobertos por um conjunto de arestas;
- contração de conjunto de arestas;
- contração de um subgrafo;
- obtenção do subgrafo induzido por um conjunto de vértices ou de arestas;
- rotina para verificar se um grafo é subgrafo de um outro;
- rotina para verificar se um conjunto de arestas é de corte;
- rotina para obtenção de uma árvore (ou floresta) geradora;

- rotina para obtenção de uma componente qualquer do grafo;
- testes para verificar se um grafo é simples, conexo, bipartido ou completo;
- obtenção de um caminho de comprimento mínimo entre dois vértices;
- obtenção de um caminho a partir de um passeio.

Para grafos dirigidos deve ser possível obter o conjunto de arestas que entram ou saem de um vértice (ou de um conjunto de vértices) ou ainda o conjunto de vértices atingíveis a partir de um dado conjunto de vértices. Deve ser possível trabalhar com grafos com pesos nas arestas ou nos vértices. Alguns algoritmos utilizam também *marcações* nos vértices ou arestas. Deve ser possível obter vértices ou arestas com peso ou marca máximos (mínimos).

Cuidado: hora de pensar *por objetos*

Após o contato inicial com o software a ser desenvolvido poderíamos ter continuado o projeto da biblioteca fixando os tipos (conjuntos, seqüências, grafos, vértices, arestas e outros que aparecessem pelo caminho) desejados e a lista de operações associadas, passando então a pensar nas estruturas de dados e finalmente programando cada uma das funções da biblioteca em uma linguagem orientada a objetos, utilizando classes para associar tipos a operações. Será isso *programar por objetos*? Acreditamos que não. Chegamos a nos enveredar por este caminho, mas percebemos que estávamos pensando no projeto da mesma forma que sempre fizemos, ou seja, isolando as rotinas de interesse e tentando estruturar os dados de maneira conveniente a este conjunto de funções; o projeto evoluía como se fôssemos programá-lo em C ou Pascal, apenas privilegiado por vantagens cosméticas como a possibilidade de definirmos duas funções de nome *insere*, uma para o tipo conjunto e outra para o tipo grafo. Estávamos confundindo certas características importantes do paradigma — ou melhor, de alguma linguagem associada ao paradigma — com o uso do próprio paradigma; *overloading* de identificadores ou operadores contribui para a qualidade do software tornando-o mais legível, mas certamente não define uma disciplina de programação.

Passamos a olhar para o problema a procura de objetos, classes e herança. Nosso *problema* era na verdade uma coleção de entidades que deveriam ser retratadas como *caixas pretas*; queríamos entregar ao usuário unidades com a qual ele pudesse interagir apertando os botões corretos, sempre prevendo a possibilidade de adaptação das *caixas* para um uso especial. Mas ao mesmo tempo queríamos que as decantadas vantagens do paradigma se refletissem também sobre o nosso trabalho enquanto implementadores, ao invés de oferecer lucros apenas para os usuários da nossa biblioteca. Tínhamos a impressão de que, sendo o software a ser desenvolvido razoavelmente grande, seria possível diminuir o nosso trabalho de programação se encontrássemos uma relação conveniente para as unidades a serem desenvolvidas.

Voltando ao domínio da aplicação, identificamos alguns objetos presentes em/um programa em grafos: vértices, arestas, grafos, subgrafos, conjuntos, etc. Tentamos

então esquematizar que relação teriam esses objetos no software e chegamos a um diagrama complicado, que hoje consideramos bizarro e sem sentido. Uma parte do diagrama é reproduzida na figura 5.1.

Figura 5.1: Parte de um diagrama inicial para modelagem dos objetos do sistema

Acompanhe nossos problemas de conceituação que geraram tal hierarquia:

- Ainda não conseguíamos uniformizar o uso de derivação. As arestas de nosso grafo de hierarquia expressavam idéias distintas, por exemplo, a aresta que rotulamos com α indicava “*vértice é um elemento do sistema*” e a aresta β “*grafo é composto por conjunto de vértices*”.
- As classes *vértices* e *arestas* se confundiam com os demais elementos do sistema, pois por exemplo um *conjunto* ou um *grafo* poderiam ser utilizados como *vértices*. Se pensamos na identidade de um vértice no sistema percebemos que um objeto não é criado para ser vértice; este é apenas um *status* que um objeto qualquer do sistema assume ao fazer parte de um conjunto utilizado para definição de um grafo. Veja que um vértice de um grafo pode ser uma aresta em outro grafo. Dessa forma, retiramos as classes *vértices* e *arestas* da nossa abstração. Dois conjuntos e mais uma função de incidência bastarão para definir um objeto *grafo*, que poderia ainda aparecer no software através de uma matriz $M_{VG \times AG}$.
- Note a confusão criada em nosso diagrama a fim de separar em categorias os grafos *com* e os *sem* pesos. Repetíamos construção análoga para representar

grafos dirigidos (e depois dirigidos com peso, dirigidos sem peso, etc). Como grafos com ou sem peso não representam objetos essencialmente distintos a melhor forma de trabalhar com pesos ou marcas em vértices ou arestas é permitir que funções sejam associadas a um grafo; dessa forma marcar os vértices de um grafo ou associar pesos a arestas significa definir uma função e associá-la ao grafo. Nosso sistema terá então uma classe nova para manipular *funções*, da qual pesos e marcas serão um caso particular. A modelagem poderia deixar de envolver classes de grafos distintas para tratar peso, mas talvez seja conveniente continuar com a classe para grafos com peso explícita em nossa hierarquia a fim de denotar uma diferenciação de operações disponíveis na interface. Apenas para os grafos com pesos associados deve ser possível utilizar a operação de *obtenção de uma árvore geradora de custo mínimo*, por exemplo.

Quanto aos grafos dirigidos, basta manipular o par ordenado associado a uma aresta ignorando a ordem para que estejamos manipulando um grafo não dirigido, ou seja, seria possível tratar a classe *grafo* como derivada da classe *grafo dirigido*. Preferimos utilizar a derivação em outro sentido: como todo grafo dirigido é um grafo, os modelaremos como especialização da abstração representada em *grafo*. Bastará redefinir algumas funções de maneira a levarem em consideração a direção das arestas para que tenhamos definida a unidade correspondente aos grafos dirigidos.

Após considerações dessa espécie pudemos simplificar a nossa hierarquia para as classes da biblioteca, como mostramos na figura 5.2.

Figura 5.2: Parte da hierarquia para as unidades da biblioteca

5.1.3 Objetos têm nomes?

Por algum tempo nos preocupamos com a identidade de um objeto, buscando uma maneira de associar a cada objeto um atributo que o distinguísse no sistema. Nossa primeira idéia foi manipular uma associação de nomes a objetos. Dessa forma, o usuário (ou o nosso sistema, automaticamente) deveria fornecer um nome a cada conjunto, grafo ou seqüência utilizada. Em meio as dificuldades para gerenciamento desses nomes (homônimos são permitidos, ou seja, um conjunto pode ter o mesmo nome que um grafo? qual a relação entre o nome de um objeto na execução e o identificador que o representa no programa?) nos demos conta de que sob o ponto de vista de execução os objetos não se destacam propriamente pelo valor que representam e sim pelo uso que é feito deste valor pelo programa. Já que nem todo valor que aparece na execução será usado de maneira especial pelo usuário não faz sentido a obrigatoriedade de associação de um nome a cada valor. O usuário poderá associar um nome a um valor toda vez que julgar conveniente: basta definir uma variável com o tipo desejado e atribuir-lhe o objeto em questão. Com essa concepção duas variáveis distintas armazenando o mesmo valor são encaradas como objetos *distintos*, embora iguais. A despeito do que possa parecer inicialmente *não* estamos utilizando para identidade de um objeto sua área de armazenamento, pois – como veremos – será possível associar várias estruturas de dados a um mesmo valor.

Na nossa aplicação o que importa é saber discernir se dois elementos do sistema são iguais, e isso pode ser feito comparando os valores armazenados. Provavelmente a questão da identidade de objetos se torna crítica em algumas aplicações, por exemplo em sistemas de processamento distribuído onde detectar se dois objetos são o *mesmo* é mais do que verificar se eles retratam valores equivalentes do sistema.

5.1.4 A questão da estrutura de dados

Para muitos sistemas a escolha de *uma* estrutura apropriada para representação dos objetos de cada classe é suficiente para atacar o problema do armazenamento dos valores manipulados. No nosso sistema, no entanto, não é possível definir uma forma de armazenamento conveniente no caso geral, pois a eficiência de cada estrutura de dados deve ser analisada em função do tipo de manipulação executada no objeto.

Um conjunto pode ser armazenado, por exemplo, através de uma lista ligada. Se existir uma relação de ordem entre os elementos do conjunto pode ser mais conveniente utilizar uma árvore de busca binária para representá-lo (o que tornaria a operação de obtenção de um elemento mínimo do conjunto mais eficiente). Nos casos onde o teste de pertinência é uma operação muito utilizada uma forma adequada para armazenamento do conjunto seria uma *tabela de hashing*.

Para grafos a situação se repete: dependendo da operação de destaque no processamento pode ser feita uma escolha diferente entre as várias possibilidades de

armazenamento (lista de adjacência, lista de incidência, matriz de adjacência, matriz de incidência).

Nós concluímos que a estruturação hierárquica para os elementos do software é muito conveniente para o projeto de sistemas onde muitas formas de implementação devem estar disponíveis, como veremos na seção 5.1.8.

5.1.5 Implementando uma lista genérica

A fim de entrarmos em contato com a linguagem C++ começamos a trabalhar com a implementação das unidades. Não tínhamos ainda uma visão global apurada do sistema (a hierarquia da figura 5.2 não nos parecia apropriada) e também não tínhamos caracterizado completamente cada interface através de sua lista de operações. Deixamos pendente a fixação dos limites de atuação do sistema; por exemplo, não sabíamos se a classe *grafo* deveria lidar com cliques, conjuntos estáveis, emparelhamentos, planaridade, coloração, etc.

Iniciamos o trabalho definindo a classe conjunto. Julgávamos que a sua implementação seria imediata se nos contentássemos com uma estrutura de dados bem simples, por exemplo uma lista duplamente ligada. Se por um lado nos assustamos com a dificuldade de implementar uma abstração tão simples, por outro tivemos a oportunidade de explorar a fundo a questão de polimorfismo e obtenção de tipos genéricos.

Um conjunto deve, a princípio, armazenar elementos quaisquer. Manter este aspecto geral para conjunto pode não ser muito fácil, pois algumas informações — como o tamanho da área utilizada pelo valor a ser armazenado — são importantes para a manipulação da lista. Nós definimos então uma classe chamada *objeto* para denotar todos os elementos do sistema passíveis de armazenamento e inserimos esta classe na hierarquia de unidades do nosso sistema. A figura 5.3 ilustra parte dessa hierarquia.

A classe *objeto* não foi criada para denotar uma unidade responsável pela manipulação de valores, pois estes deveriam sempre ser criados pelas diversas subclasses de *objeto*. Na nossa modelagem *objeto* é uma classe **abstrata**, que foi introduzida no software a fim de centralizar características comuns (básicas) a todos os elementos de uma parte do sistema. Em C++ não há uma maneira “limpa” de especificar uma classe como sendo abstrata¹.

Observe a hierarquia para conjuntos na figura 5.3. Começamos com a classe *conjunto* e a especializamos para armazenar classes de valores homogêneos. É natural começarmos a implementação a partir da classe mais genérica. O primeiro passo seria

¹Uma possibilidade seria impôr restrições sobre a visibilidade dos construtores da classe, mas isto pode implicar em algumas restrições ao desenvolvimento de rotinas para classes fora dessa hierarquia que queiram utilizar um *objeto* como argumento.

Figura 5.3: A classe *objeto* e suas subclasses

pensar em algo como:

```

classe conjunto : public objeto
{
    lista-ligada* l;           // apontador para lista ligada do conjunto
    tam t;                     // armazena o número de elementos do conjunto

    public:
        conjunto();           // Rotina utilizada para criação de conjuntos
        conjunto(conjunto&); // Rotina utilizada para criação de conjuntos
                               // a partir de outro (inicialização)
        ~conjunto();          // Rotina para destruição de conjuntos (chamada
                               // automaticamente quando um objeto sai de escopo)

        conjunto& operator=(conjunto&);
                               // Atribuição
        :
        conjunto& insere(objeto&);
                               // Inserção de elemento no conjunto
        :
}
  
```

Observe que a rotina de inserção foi definida para receber qualquer *objeto* do sistema como argumento. A implementação dessa função deve verificar se o objeto a ser inserido já não aparece no conjunto; para isso deverá percorrer a lista ligada comparando cada elemento armazenado com o elemento a ser inserido. O problema é como fazer essa comparação, já que os valores são de classes quaisquer do sistema, podendo cada valor ser de uma espécie diferente. Mesmo que para cada uma das classes esteja definido um operador `==` (na verdade definido para a classe *objeto*, e redefinido em cada uma das especializações dessa classe genérica), não há como implementar em C++ a rotina de comparação de maneira a levar em consideração o tipo. Vejamos a seguir como o problema se manifesta na hora de definirmos esses operadores.

A declaração

```
objeto::operator==(objeto&) { ... }
```

define que cada objeto do sistema é dotado de operador para comparação de igualdade. A rotina *default* associada a esse operador é curiosa, pois como não faz sentido comparar elementos gerados pela classe *abstrata* sua utilização deveria indicar situação de erro. Assim, colocamos a operação na interface de *objeto* mais proibimos que a rotina seja utilizada a fim de garantir que só especializações de *objetos* sejam comparadas.

Quando implementamos a classe dos reais o operador de igualdade deve ser redefinido de maneira a comparar as áreas de armazenamento utilizadas para objetos da classe *real*:

```
real::operator==(objeto&) { ... }
```

Embora só faça sentido comparar reais com objetos da mesma classe (ou de suas subclasses), não há em C++ como definir `real::operator==(real&)` como especialização para `objeto::operator==(objeto&)`.

A rotina genérica envolvendo dois objetos da classe *objeto* pode ser especializada através do mecanismo de *virtual*. Isso significa que se temos a declaração “*real* r1, r2;” a expressão `r1==r2` será tratada como `r1.operator==(r2)` e entre as várias funções `operator==` definidas será utilizada a correspondente à classe de r1.

A idéia de *virtual* é vincular a cada objeto o comportamento especial para sua classe. Toda a descrição de uma classe funciona como um conjunto de informações “carregadas” por todos os objetos, mesmo as funções estão atreladas a ele já que só podem ser utilizadas em um contexto que envolve algum objeto definido para a classe. Assim, o mecanismo só leva em conta o objeto “dono” da manipulação, ou seja, o primeiro argumento do uso do operador de comparação.

Como não é possível *virtualizar* os parâmetros das funções não há como implementar `real::operator==(objeto& o)`, pois *o* pode tanto ser um real como um grafo.

Concluimos então que o mecanismo para definição de rotinas *virtuais* em C++ não consegue tratar completamente a questão de informação sobre o tipo de um objeto durante a execução, ou melhor, não consegue lidar com essa informação quando mais de um objeto está envolvido na manipulação. A solução seria fazer com que o próprio sistema gerenciasse a manutenção de informação extra sobre o tipo de um objeto.

Para resolver nossos problemas de investigação de tipos bastaria dotar cada classe com operações do tipo *is-classe*. Por exemplo, um elemento criado pela classe *racional* devolveria verdadeiro como resultado das operações *is-objeto*, *is-real*, *is-racional* e falso para coisas do tipo *is-conjunto*, *is-grafo*. Isso poderia ser conseguido se especificássemos na interface das classes uma forma de “auto-avaliação”, ou seja, expandíssemos o comportamento esperado de um objeto de maneira que se pudesse obter respostas às perguntas do tipo “o objeto é da classe *tal*?”. Observe que se trata de uma coleção enorme de operações a serem definidas. Uma implementação possível para essa idéia seria adicionar à interface de cada classe *X* da hierarquia correspondente a *objeto* uma nova função; isto pode ser feito alterando apenas as definições de *objeto* e *X*:

```
classe objeto
{
    :
    public:
        :
        virtual boolean is-X () { return false; }
        :
}

classe X: objeto
{
    :
    public:
        :
        boolean is-X () { return true; }
        :
}
```

Dessa forma teríamos garantido o teste *is-X* para todas as classes derivadas de *objeto* e somente para objetos da classe *X* (e de suas subclasses) teríamos uma resposta afirmativa para o teste. É claro que essa solução é inconveniente, pois a cada nova classe acrescentada ao sistema alteraríamos a interface da classe *objeto* e, conseqüentemente todo o sistema.

Colocando a informação de tipo em um objeto

Programas tradicionais em C ou Pascal já lidavam com o problema de associar um tipo à uma área de armazenamento em tempo de execução. Para isto basta acrescentar um campo à definição do valor e codificar nesse campo a informação de tipo desejada. Ou seja, cada uma das classes poderia ser identificada por um número, que estaria presente em cada um dos objetos da classe de maneira a identificar seu tipo. De posse desse tipo de informação, a implementação do operador de comparação poderia ser feita da seguinte forma:

```
boolean real::operator==(objeto& o)
{
    if (this ->tipo() == o.tipo()) // Compara o tipo dos dois
                                    // objetos envolvidos na operação
    {    ...    }
    else return false;
}
```

Esta solução parece nos levar de volta à situação anterior à utilização de tipos abstratos de dados e herança, pois perdemos a ligação entre um valor e seu tipo e tentamos resgatar essa informação através de valores em área de armazenamento.

Nós resolvemos no nosso sistema a questão de identificação de tipo de uma maneira mais próxima à filosofia de programação por objetos. Pelo menos na nossa solução não estão presentes os principais problemas ocasionados pela manutenção de um campo para codificação do tipo, ou seja:

- o desperdício de memória (pois em cada objeto é repetida uma informação referente à uma classe de objetos);
- dificuldade para extensão do sistema (um usuário que queira expandir o sistema deverá de alguma forma conhecer o restante da hierarquia, ou pelo menos a forma de atribuição de códigos a cada uma das classes).

Nós nos baseamos nas idéias de *instância* e *método de classe* do *Smalltalk*, implementando-as em C++ através de *static member*. Isso significa que associamos a cada classe — e não a cada objeto — uma área de armazenamento; nesta área nosso sistema coloca um número identificador para a classe. Essa associação é feita por ocasião da primeira criação de um objeto da classe. Este esquema evita a utilização de área extra para cada objeto, mas continua causando problemas a um usuário que queira acrescentar novas classes ao nosso sistema, pois sem conhecer a técnica que utilizamos para codificação dos tipos não teria como dar continuidade ao processo de identificação dos objetos. Embora nossa técnica para associação de números à classe seja bem simples (apenas acrescentamos duas linhas ao construtor de cada

classe) fizemos um utilitário que dada a definição de uma nova classe acrescenta-lhe os comandos para controle, tornando transparente ao usuário da biblioteca a forma de codificação para tipo e permitindo que, dados dois objetos quaisquer, se decida se eles pertencem a mesma classe ou não.

Se eliminamos a possibilidade de coleções polimórficas do nosso sistema temos que o mecanismo de funções virtuais é suficiente para o gerenciamento de informação de tipo durante a execução.

5.1.6 Implementando tipos parametrizados

Uma variação da idéia de lista genérica é a definição de coleções homogêneas para um dado tipo T ; por exemplo a classe *conjunto- T* denotaria a classe *conjunto* especializada de forma a armazenar apenas elementos do tipo T . No caso específico de conjuntos, uma maneira de definir tais classes seria especializar a operação *insere*, permitindo a inserção de elementos apenas para uma dada espécie. O problema na programação da rotina *insere* seria, novamente, a necessidade de conhecimento extra acerca do argumento de uma função membro. Preferimos definir um utilitário para criação de classes paramétricas: dada a definição de uma classe em função de um identificador T , o utilitário gera uma instanciación dessa classe genérica substituindo as ocorrências do tipo T pelo tipo desejado. Em [Str86a] Stroustrup sugere uma geração análoga de classes paramétricas através da definição de macros.

5.1.7 Reinventando a roda

Observe que nas seções anteriores a discussão focalizou aspectos gerais da implementação de coleções de elementos genéricos. Por algum tempo nosso interesse foi desviado do sistema a ser desenvolvido para a questão de polimorfismo e tipos genéricos em C++; era como se estivéssemos incumbidos de programar uma biblioteca com tipos básicos para a programação em geral.

A impressão que temos agora — ao analisarmos o rumo que nosso desenvolvimento de programas assumiu — é que estávamos em busca de um ponto onde o paradigma dos objetos manifestasse a sua força em separar implementação de especificação; não nos importávamos portanto em *começar do zero*, implementando as formas usuais de estruturas de dados a fim de formar uma base para o desenvolvimento do sistema. Assim, definimos classes para listas, listas ligadas, listas duplamente ligadas, árvores binárias e tabela de hashing. Resolvemos também implementar as classes responsáveis por pilhas e filas, fornecendo para cada uma dessas estruturas de armazenamento várias formas de implementação.

Aproveitamos a atividade de programação envolvida na construção de cada uma

dessas estruturas para melhor compreender a linguagem C++ (muitas vezes nos perdendo em alguns pontos obscuros de sua utilização). Não nos contentamos em construir as classes relativas a estruturas de dados básicas e resolvemos oferecer também classes para inteiros, reais, racionais, complexos e cadeias de caracteres. Trabalhando com estas espécies — bem simples — de valores pudemos estudar em detalhe uma questão importante do desenvolvimento em C++: quando utilizar funções *member* e quando utilizar funções *friend*.

Em C++ as funções definidas como parte da interface de uma classe são chamadas de funções *member*, pois de uma certa forma são de propriedade dos objetos da classe: a utilização de uma função *member* sempre se dá em cima de um dado objeto (que funciona como um parâmetro implícito para a função). Uma função declarada como *friend* de uma classe *C* denota uma rotina que não está atrelada a classe alguma, mas foi declarada como tendo o privilégio de acessar a estrutura interna da classe *C*. Observe o exemplo:

```
classe X
{
    int i;
    :
    public:
        int fx();
        friend int g (X&, Y&);
};
```

A função *fx* é *member*, portanto o seu uso tem sempre a forma *x.fx()* ou *px —>fx()*, onde *x* é um objeto da classe *X* e *px* um apontador para um objeto da classe *X*. A função *g*, sendo *friend*, recebeu passe livre para manipulação de objetos da classe *X* e tem sintaxe de utilização semelhante a de qualquer função em C (sem nenhuma obrigação de receber objetos da classe *X* como parâmetro). Tanto *fx* quanto *g* podem acessar livremente a parte privada *i*.

Nós encarávamos a presença de funções *friend* na linguagem como uma brecha para a má utilização das idéias de tipos abstratos de dados. Os primeiros artigos de Stroustrup sobre C++ (que tratavam da extensão da linguagem C) também demonstravam uma certa resistência à idéia de *friend*; se permitia a declaração de classes *friend* — o que acaba fundindo a implementação de duas unidades — mas não se julgava conveniente estender o *status* de *friend* para funções. O fato de Stroustrup utilizar durante todo o livro que descreve C++ *friends* para implementar operadores aritméticos atiçava nossa curiosidade sobre o mecanismo. Consultamos outras referências sobre a linguagem e sua utilização para programação por objetos e então percebemos que não éramos os únicos a ter dúvidas sobre a existência de uma razão forte para a presença de funções *friend* na linguagem. Em [Wie 87] justificava-se a utilização de *friend* observando-se que “funções com dois parâmetros não podem ser

implementadas como *member*”, o que não é verdade. Uma função *member* pode manipular livremente os elementos da classe, não precisando se restringir ao parâmetro implícito em sua utilização.

Em relação a utilização de funções *member* e *friend*, concluímos que:

- funções para as quais se julga conveniente conversão de tipo implícita devem ser declaradas como *friend*:

A passagem de argumentos (explícitos) a uma função envolve a chamada de operações definidas para conversão de tipos (como em Pascal, por exemplo, onde a função de exponenciação *exp* foi definida para lidar com argumentos reais mas podemos utilizá-la com argumentos inteiros, como em *exp(2)*). Considere como exemplo as seguintes definições:

```
classe A
{
    :
    public:
        A();           // construtor para a classe A (chamado automaticamente
                       // durante a criação de qualquer elemento da classe A)
        A(int&);       // construtor que cria um objeto da classe A
                       // a partir de um inteiro
        A(X&);         // construtor que cria um objeto do tipo A a partir
                       // de um elemento do tipo X
    :
}

:
void f (A&);
:
```

Ao utilizarmos $f(x)$ (x da classe X) o compilador identifica a função f como tendo um elemento do tipo A como parâmetro. Como está definida a conversão de X para A será executada a chamada $f(o)$, onde o é um objeto temporário criado através da rotina $A(X\&)$: Se f tivesse sido definida como *member* a chamada $x.f()$ resultaria em um erro de compilação após buscar-se a operação f na interface de X ;

- operações que alteram o objeto recebido devem ser declaradas como sendo *member*, o que garantiria que o argumento da função é realmente um objeto real (e não um objeto temporário criado para conversão de tipos);

Assim, é razoável que operadores aritméticos sejam definidos como funções *friend*, pois desta forma seria possível a conversão implícita de tipos nas expressões.

A desvantagem de desvincular dessa forma a *função* da *classe* é que não há como especializar a função na hierarquia usando *virtual*.

Consideramos que a fase em que deixamos de lado a biblioteca de grafos a fim de nos dedicarmos à implementação das classes básicas foi produtiva, pois serviu para que nos acostumássemos a olhar para a funcionalidade a ser oferecida de maneira a captar o contexto mais geral possível para uso. No entanto, não achamos que essa carga de trabalho básico seja conveniente para um primeiro contato com o paradigma. O projeto de uma coleção de classes com o intuito de servir de base para a programação tem uma responsabilidade muito grande, já que imperfeições na modelagem das classes básicas poderiam vir a comprometer seriamente o restante do sistema. Como não julgávamos confiável o produto de nossa primeira experiência com programação por objetos e com C++ tínhamos a sensação de termos dispendido esforço de maneira desorganizada e até mesmo desastrosa.

Recentemente confrontamos nossa especificação de classes básicas com a biblioteca de classes do compilador g++[gnu]. Se por um lado nos confortamos ao perceber que muitas de nossas opções de implementação coincidem com as adotadas pelo sistema, por outro lamentamos não ter começado nosso trabalho pelo estudo de tais implementações *reais*. A descrição das classes definidas em libg++ destaca esquemas importantes de implementação, ou seja, serve de guia para uma linha inicial de divisão dos dados em categorias. Por exemplo, na documentação de libg++ encontramos as funções *friend* classificadas como *construtivas* (pois não modificavam o objeto recebido para manipulação) e as *member* como *destrutivas*; dessa forma um novato pode perceber através de categorias de uso a que tipo de coisa cada uma das formas para definição de uma operação se presta. As coleções definidas eram todas homogêneas e existe um utilitário — como no nosso sistema — definido para especificação de novos tipos a partir da instanciação de um dado tipo parametrizado. Já a forma para especificação de *iterator* (mecanismo para percorrer os elementos de uma coleção) oferecida pela biblioteca libg++ era muito mais conveniente que a nossa.

5.1.8 Tratando as alternativas para armazenamento de um objeto

Voltando ao problema da implementação de várias formas de armazenamento disponíveis para representação de elementos de uma classe, investigamos até que ponto os conceitos de objetos eram necessários para a obtenção de independência entre interface e implementação. Mais especificamente, nosso objetivo era descobrir se *he-*

rança contribuía de forma determinante para a separação entre uso e codificação de operações ou se tipos abstratos e encapsulamento seriam suficientes.

As possíveis representações para a classe *conjunto* poderiam ser expressas facilmente através da seguinte hierarquia:

Dessa forma a classe *conjunto* seria abstrata, pois objetos do tipo *conjunto* deveriam ser criados já atrelados à forma de representação escolhida. Encontramos um inconveniente para essa hierarquia: fica difícil especializar a classe *conjunto* levando em consideração o tipo do elemento a ser armazenado. Se quisermos acrescentar uma classe para manipular conjuntos de inteiros ou conjuntos de conjuntos onde a encaixaríamos? E como colocar na hierarquia as diversas formas de implementação para tal classe? Outro problema é que nem todos os conjuntos se prestam a todas as representações possíveis; por exemplo, apenas conjuntos cujos elementos apresentem relação de ordem total poderão ser representados por uma árvore de busca binária. Utilizar a mesma hierarquia para representar o *comportamento* do tipo *conjunto* e suas *formas de implementação* mistura níveis distintos de abstração.

Em nosso sistema separamos totalmente a classe *conjunto* das formas disponíveis para armazenamento:

A classe *conjunto* fica sendo composta de uma representação e de operações disponíveis. Um objeto *conjunto* armazena um apontador para um objeto da classe *repr*. Descrevemos essa relação entre as unidades da seguinte forma:

```

class conjunto
{
    repr* pr;
    :
    public:
        void insere(elemento&);
        :
}
  
```

Dessa forma a função de inserção, por exemplo, poderia ser definida da seguinte forma:

```

void insere::conjunto (objeto& o) { pr —>insere(o); }
  
```

A própria representação correspondente a *pr* seria responsável pela execução da rotina *insere* apropriada. A escolha da forma desejada para implementação seria feita através da passagem de um argumento para a criação do objeto, ou seja, definindo um *construtor*² cujo parâmetro definisse o tipo desejado para a representação.

Vejamos se poderíamos expressar essa idéia para conjunto sem definir uma hierarquia de representações: definiríamos um módulo (classe) independente para cada uma das representações possíveis, de maneira que o mesmo conjunto de rotinas estivesse definido em cada um destes módulos:

A implementação das rotinas do módulo conjunto se utilizaria das funções externas cujos identificadores seriam *remove*, *insere*, *busca*. Para obtenção do código objeto

²Um construtor é uma rotina executada toda vez que um objeto da classe é alocado, tanto no caso de criação explícita (através da declaração de uma variável do tipo da classe ou pela utilização do operador *new*) quanto na criação implícita (por exemplo, um objeto criado automaticamente para receber o valor de retorno de uma função ou para armazenar o resultado de uma conversão de tipos).

referente a uma dada implementação bastaria juntar (na fase de *link*) a classe conjunto ao módulo que define a representação desejada. Como são utilizados os mesmos identificadores nos vários módulos apenas uma das representações poderia ser utilizada de cada vez, o que impediria a manipulação em um mesmo programa de conjuntos com diferentes representações. Dessa forma, consideramos a possibilidade de definição de uma hierarquia de representações fundamental para possibilitar o fornecimento de alternativas de implementação para um objeto.

A hierarquia de nosso sistema para implementações se baseia na relação entre as unidades que reproduzimos na figura 5.4.

Figura 5.4: Hierarquias distintas para conjunto e formas de implementação

Observe que ao especializarmos o conjunto generalizamos a possibilidade de armazenamento.

5.1.9 Múltiplas representações para um mesmo objeto

Poderiam ser definidas rotinas para transição entre representações e teríamos assim a possibilidade de associar diversas estruturas de dados a um conjunto durante a

execução. Isto poderia ser conveniente para algumas seqüências de operações. Por exemplo, se são realizadas k operações de obtenção de mínimo e depois k testes de pertinência poderíamos utilizar duas estruturas de armazenamento, inicialmente uma apropriada para encontrar o mínimo e depois uma conveniente para busca de elementos:

```
conjunto C(Bin);           // cria conjunto representado por uma árvore de busca binária
...
C.minimo();                 // obtenção de elemento mínimo de C
...                         // algumas inserções, remoções, etc.
C.minimo();
C.minimo();
...
C.minimo();
C.muda-repr(Hash);         // Transforma a representação de C
                           // em uma tabela de hashing
... = C.pertence(x);        // Verifica se  $x$  pertence a C
... = C.pertence(x);
:
:
```

Poderíamos atrelar a um objeto simultaneamente várias formas de representação, bastaria associar ao objeto uma lista ligada de representações e definir um mecanismo que mantivesse consistência entre as diversas formas de armazenamento.

Nenhum esforço de implementação seria necessário se atualizássemos simultaneamente todas as representações associadas ao objeto, pois bastaria percorrer a lista ligada de representações aplicando a manipulação a cada elemento dessa lista. Essa manutenção conjunta de várias representações poderia — por incrível que pareça — resultar em alguma economia de processamento; por exemplo, para um conjunto onde o número de inserções e remoções é irrisório e um grande número de operações *mínimo* e *pertence* entremeadas são realizadas no conjunto pode valer a pena o duplo trabalho de atualização.

Outra possibilidade para manutenção da consistência de representações é manter informações sobre as operações realizadas, marcando para cada representação qual seu nível de atualização. Assim, das várias representações disponíveis teríamos a cada instante uma atualizada e as demais “em compasso de espera”, podendo ser ativadas a qualquer momento.

5.2 Considerações finais sobre o projeto

Após trabalharmos com a implementação das unidades responsáveis por conjuntos (lidando com coleções heterôgeneas e explorando a questão das múltiplas representações) voltamos a olhar globalmente para a biblioteca de grafos. Resolvemos “pôr os pés no chão”: tratávamos apenas conjuntos homogêneos e em todos os nossos testes usávamos listas com dupla ligação como forma de armazenamento para conjuntos. Tomamos essa decisão após nos convenceremos de que poderíamos estender o sistema de modo a englobar os aspectos “avançados” se quiséssemos; como já sabíamos de que maneira seria possível implementar tais estruturas poderíamos continuar o desenvolvimento nos restringindo aos casos mais simples (o que facilitaria a depuração dos programas), sem temer inconsistências finais entre as nossas especificações e as questões de classes genéricas e múltiplas representações simultâneas.

Passamos a especificar a classe *grafo*, supondo prontas unidades que manipulassem matrizes, funções, seqüências. Não havia dificuldade na definição das operações, pois em geral executá-las significava chamar operações sobre as estruturas de armazenamento. Como a especificação da classe *grafo* era feita em *alto nível*, sem tentar extrair vantagens da forma de armazenamento — que sequer conhecíamos —, não existiam propriamente dificuldades na programação das rotinas. Mas, a cada rotina que especificávamos, uma dúvida nos atingia: *quem fica com esse trabalho?* Sabíamos que se passássemos o trabalho para a classe responsável pelo armazenamento obteríamos rotinas mais eficientes.

Ilustraremos essa espécie de *dilema* através da especificação da operação de *remoção de um vértice do grafo*. Uma forma razoável para denotar essa operação é $G - v$, ou seja, estamos redefinindo o operador binário “ $-$ ” de forma que seu primeiro operando seja um grafo, o segundo um vértice³ e o resultado da operação um novo grafo:

grafo& grafo::operator—(vertice&)

Ao codificarmos essa operação deveremos decidir entre programá-la como uma seqüência de manipulações que resulta no grafo esperado ou passar o trabalho para a sua representação.

³Na seção 5.1.2 nós argumentamos pela extinção da classe *vértice* como parte do sistema (figura 5.2), pois estes elementos do software não eram uma abstração importante para o projeto das unidades. Mais tarde, porém, resolvemos que a possibilidade de criar uma variável do tipo *vértice* que armazenasse um elemento de um conjunto utilizado em algum grafo tornaria a utilização das rotinas de nossa biblioteca em algoritmos programados pelos usuários mais legível. Enfim, uma mudança na modelagem que influencia apenas a visão que os usuários têm das classes.

Supondo a seguinte especificação para um grafo

```
classe grafo : objeto
{
    repr* pr;
    conjunto* VG, AG;
    :
    public:
        grafo& operator—(vertice&);
}
```

poderíamos definir a operação de remoção de vértice simplesmente como

```
grafo& grafo::operator-(vertice& v)
{
    return new grafo(pr —>remove_vertice(v));
}
```

Dessa forma, se a representação adotada para o grafo no momento de sua criação for uma lista de adjacência esta executará sua rotina para remoção das ocorrências de v , devolvendo uma nova lista de adjacência que será utilizada para criação do grafo resultante da operação. Analogamente, se pr for um apontador para uma matriz de adjacência a rotina remove-vértice para este tipo de representação devolverá a matriz resultante da eliminação da linha e coluna correspondentes a v .

Observe então que se tomamos o cuidado de dotar todas as possíveis representações com uma dada operação temos que a implementação da rotina correspondente na classe grafo é imediata e resulta numa forma apropriada para manipulação da estrutura de armazenamento de modo a obter o resultado desejado para a operação.

Por outro lado, poderíamos ter optado em não fornecer para cada uma das representações a operação remove-vértice. Poderíamos então executar a operação desejada através de operações envolvendo grafos:

```
grafo& grafo::operator—(vertice& v)
{
    conjunto auxvert, auxarest;
    // conjuntos auxiliares
    função f; // variável a que associaremos a função de incidência do grafo

    f = this —> função-incid();
    // obtém a função associada ao grafo,
    // ou seja, a partir da representação do grafo será obtida a função
    // de incidência. No fundo, uma tradução de representação.
```

```

auxvert = VG - v;
auxarest = G.estrela(v);
           // obtém o conjunto das arestas com uma ponta em v
auxarest = AG - auxarest;
           // elimina do conjunto de arestas as que têm
           // uma ponta em v
f.restringe(auxarest);
           // restringe o domínio de f ao novo conjunto de arestas

return grafo (auxvert, auxarest, f);
           // constrói para valor de retorno do operador
           // o grafo a partir da definição do conjunto de vértices, arestas
           // e função de incidência
}

```

Acreditamos que a nossa especificação consegue expressar as abstrações relacionadas a algoritmos em grafos realmente separando o conceito matemático do elemento de sua forma de armazenamento durante a execução. O preço a ser pago pela “aparência sedutora” é bem alto, pois objetos temporários e auxiliares são alocados constantemente (envolvendo muito espaço de memória) e a execução da maioria das operações requer muitas buscas em tabelas, consumindo assim bastante tempo de processamento.

5.2.1 Estágio atual do software

Utilizamos no desenvolvimento dos nossos programas o compilador Zortech, versão 1.5, processando em um microcomputador de 8 MHz e 704 Kbytes de memória RAM. No apêndice A descrevemos uma parte da especificação das classes disponíveis na biblioteca.

Sobre o estágio atual do software, cumpre ressaltar dois pontos importantes:

- Ignoramos as questões de entrada e saída
Em nossos testes utilizamos as rotinas de entrada e saída mais rudimentares possíveis. A leitura era feita através de uma lista de triplas (ou pares, e nesse caso associávamos números inteiros as arestas descritas pelo par); a partir da lista construíamos os conjuntos de vértices e arestas e a função de incidência (a partir da qual podíamos obter qualquer uma das outras representações). Também permitíamos entrada através da matrizes de incidência do grafo. Consideramos a questão de entrada de objetos como conjuntos, vértices ou funções difícil de ser atacado de uma maneira conveniente. Certamente essa parte do sistema deve ser desenvolvida a partir de uma biblioteca básica para interfaces. Nossa saída de grafos é feita através de sua matriz de incidência. O desenho de um grafo é uma questão mais delicada, que deveria envolver interação com

o usuário. Em [GNV 88] encontramos a descrição de um programa que desenha grafos dirigidos a partir de uma entrada padrão que poderia ser facilmente construída pelas nossas rotinas.

- **Nosso sistema não funciona**

Uma parte do código escrito provavelmente não funciona corretamente, pois sequer foi devidamente testada. Para piorar, existem rotinas que não compilam e o erro apontado indica um problema sério de projeto: ambigüidade na interface. Simplificando, esse problema costuma aparecer na seguinte situação:

```
classe A { ... };
classe B: public A { ... };
classe C: public B { ... };

overloading f;
    // Indica que o identificador f
    // denotará mais de uma função
void f(A&) { ... rotina  $f_a$  ... };
void f(B&) { ... rotina  $f_b$  ... };

A va;
B vb;
C vc;

f(va); // Chama rotina  $f_a$  associada a f
f(vb); // Chama rotina  $f_b$  associada a f
f(vc); // Erro: Ambigüidade
        // poderia ser tanto f(A&) quanto f(B&)
```

Mas, mesmo com alguns problemas, não consideramos o projeto simplesmente inacabado. Houve uma mudança de postura durante o desenvolvimento: não buscávamos mais um produto e sim um *protótipo* que permitisse a programação de um sistema de qualidade. Assim, a atividade de *projeto* do sistema está terminada. Sabemos quais unidades devem ser programadas e que aspecto final assumirá o sistema. Falta preencher as lacunas, que no caso são algumas as rotinas a serem implementadas e pequenas alterações nas listas de parâmetros definidas para as operações.

5.2.2 O projeto como aprendizado

Não há dúvida de que foi *programando* que passamos a realmente compreender como objetos, classes e herança aparecem no software, mas não é a consolidação de conceitos que mais nos chama a atenção no processo. Descobrimos que o estudo do paradigma fez com que criássemos alguns mitos ou ilusões que foram por água abaixo após a experiência:

- Achávamos promissora a linha de conceituação formal para derivação de tipos abstratos de dados através da relação de subtipo. Constatamos que o reaproveitamento de unidades quer servir a vários propósitos, e expressar herança como simples especialização de tipos já definidos é por demais restritivo;
- Acreditávamos que o desenvolvimento de software guiado pela idéia de hierarquia de unidades fornecia uma estrutura fácil de ser seguida durante o projeto. No entanto descobrimos que é difícil saber qual a próxima peça a ser encaixada no quebra-cabeça: não sabemos se atacamos o problema em largura (tentando enxergar o software de uma maneira mais geral) ou em profundidade (descendo fundo na implementação de cada unidade). Se queremos redefinir uma classe localizada no “meio” da árvore boa parte da especificação deve ser revista;
- Estranhávamos que os artigos publicados para descrever um software desenvolvido com programação por objetos em geral não apresentavam graficamente (através de uma árvore ou grafo orientado) a hierarquia resultante do projeto das unidades. Hoje percebemos que herança é utilizada tanto quanto possível para diminuir a quantidade de código a ser escrito, tornando assim a hierarquia carregada demais para ser apresentada aos usuários;
- Não imaginávamos que teríamos tantos apontadores e derreferenciação nos nossos programas;
- Quando começamos a especificação da classe de grafos achávamos que estávamos conseguindo uma forma simples de expressar as manipulações, mas observando a descrição de algumas bibliotecas de classes concluímos que não há uma maneira razoável de descrever as ferramentas que fornecemos. Ou seja, é possível que na maioria das vezes as classes e interfaces pareçam de fácil utilização apenas para os seus implementadores, de modo que os usuários não conseguem aproveitar a potencialidade das definições da biblioteca. Nós especificamos vários algoritmos através das rotinas de nossa biblioteca (vide apêndice B), mas talvez a facilidade com que o fizemos se deva à experiência de implementação das rotinas.

Nós tivemos muitos problemas no início do trabalho por não termos entendido satisfatoriamente⁴ o que significavam as derivações de classes, quais eram os direitos de uma classe derivada em relação a “seus pais”, etc. Não sabemos se tais confusões são comuns entre os que começam a trabalhar com o paradigma através de um ambiente muito parecido com o que utilizavam antes ou se poderiam não ter acontecido caso tivéssemos nos dedicado por mais tempo ao estudo conceitual do paradigma antes de começar a programar.

⁴Embora acreditássemos que sim

5.3 Em busca de uma metodologia

A experiência serviu para que definíssemos uma visão *pessoal* da forma de atuação das idéias de objetos no desenvolvimento de software. Acreditamos que conciliamos o estilo — e vícios — anteriores de programação à possibilidade de dividir o programa em partes que associam dados a funções. Valorizamos aspectos do desenvolvimento de programas que antes ignorávamos; por exemplo:

- desenvolver o software numa seqüências de versões, utilizando inicialmente estruturas de armazenamento bem simples, que possibilitem obter rapidamente uma idéia do tipo de sistema que estamos construindo;
- pensar no problema em termos de biblioteca, ou seja, ao projetar um programa específico abstrairmos da funcionalidade a ser oferecida características que permitam obter rotinas que não sejam totalmente configuradas para uma particular aplicação.

Observe que a aplicação das idéias acima demanda um esforço de desenvolvimento que só seria compensado se realmente conseguíssemos reaproveitar partes desenvolvidas para aplicações específicas. Assim, a utilização de programação por objetos como disciplina para desenvolvimento de software requer que o implementador se volte constantemente para o conjunto de classes e rotinas já definidas. O que podemos esperar do “banco” de rotinas reutilizáveis? Se o banco for constantemente atualizado com as classes derivadas para uso específico provavelmente teremos tantas classes que só através de um bom sistema de recuperação de informação poderíamos encontrar uma unidade apropriada; se por outro lado restringimos o depósito de novas unidades a fim de manter disponíveis apenas classes de uso geral castramos o caráter evolutivo do sistema de unidades reaproveitáveis.

A utilização de programação por objetos com o intuito de baixar os custos de desenvolvimento exigirá uma certa coerência na especificação das unidades. Embora não tenhamos experiência com desenvolvimento de software em larga escala, acreditamos que somente através de um consenso sobre o que são classes e como será utilizada herança se poderá desenvolver especificações para classes reutilizáveis. Alguns trabalhos enfocam a questão de uma metodologia para desenvolvimento orientado a objetos [Bo 86, S-S 88], sugerindo diagramas para descrição do sistema e fixando as propriedades desejadas em uma componente do software que retrate um objeto.

Embora o próprio *desenvolvimento orientado para objetos* não defina ainda uma disciplina de produção de software muitos trabalhos já traçam diretrizes para uma avaliação de estilo em um *sistema orientado* [LHR 88, L-H 89, Sak88b]. Esses trabalhos têm em comum o fato de utilizarem a *Lei de Demeter* como métrica para qualidade de programas. A Lei de Demeter é uma espécie de regra simples para se produzir “boa programação por objetos”. A regra é garantir que os métodos de uma classe não dependam de forma alguma da estrutura de outra classe que não seja sua

superclasse imediata. Além disso, cada operação deve enviar mensagens apenas para um conjunto limitado de classes. Também é recomendável minimizar a duplicação de código, o número de argumentos enviados aos métodos e o número de métodos por classe. Observe que, segundo estes princípios, a nossa biblioteca foi desenvolvida com um péssimo estilo, pois a maioria das classes dependem das unidades referentes à implementação.

Uma visão crítica do uso dos objetos como metodologia de desenvolvimento aparece em [Mos 89] e [Gut 89]. Relatos de experiências com desenvolvimento de sistemas com orientação a objetos podem ser encontrados em [HSS 89], [Bar 86] e [A-S 88].

5.4 Afinal, o que é programação por objetos?

Um dos nossos objetivos durante o estudo do paradigma de programação por objetos era encontrar uma resposta razoavelmente satisfatória para a questão “o que é programação por objetos?”. Todas as vezes em que tal pergunta nos foi dirigida ainda estávamos procurando construir uma definição e improvisávamos uma resposta traduzindo a resposta padrão “*POO = tipos abstratos + encapsulamento + herança*” para a linguagem de hierarquia, módulos e interface.

Conforme íamos nos acostumando com as idéias dos objetos consolidávamos a idéia de programação por objetos como um estilo de programação baseado em uma forma particular de encarar os elementos envolvidos na computação. Dessa forma nossa esperança em obter uma maneira conveniente para descrição do paradigma diminuía, pois acreditamos que caracterizar uma disciplina de desenvolvimento é uma atividade muito difícil.

Terminamos o trabalho sem uma resposta ideal para a pergunta original do trabalho, mas com algumas opiniões formadas sobre programação por objetos:

- *não* é uma solução para os problemas de produção de software
- *não* é expressar as entidades do software de uma forma mais próxima do mundo real
- *não* permite reutilização de software como a que ocorre com componentes de hardware
- *induz* um melhor desenvolvimento da especificação e projeto do sistema antes de se passar à programação de rotinas
- *permite* que a idéia de tipos em uma linguagem seja usada de forma mais efetiva
- *sugere* uma modularização do sistema que minimiza a duplicação de código

5.5 Outras questões interessantes

As idéias de programação por objetos se espalharam em quase todas as direções possíveis. Em nosso trabalho pudemos abordar apenas algumas das áreas onde os objetos causaram sensação. Três assuntos não presentes nesse trabalho são especialmente interessantes:

- a relação entre programação por objetos e bancos de dados [LNCS3]
Idéias oriundas de linguagens orientadas a objetos têm sido utilizadas para tratamento de banco de dados onde os elementos armazenados não apresentam uma forma usual. A ligação entre extensões do modelo relacional de dados, modelos semânticos e diagramas para hierarquia de objetos têm sido estudadas;
- programação por objetos e sistemas distribuídos [Jul 88]
O modelo de objetos parece se adequar à implementação de sistemas operacionais para ambientes distribuídos;
- o ensino do paradigma de programação por objetos [Beck89, L-R 89, K-M 88]
Há hoje em dia muita discussão acerca de maneiras apropriadas para ensinar os principais conceitos de programação por objetos. A impressão que temos é de que ensinar programação por objetos no futuro será como ensinar programação estruturada hoje: uma questão de passar para o aluno a forma de pensar nos problemas envolvendo as estruturas básicas. Mas isto só poderá ser tornar efetivo quando os objetos realmente aparecerem naturalmente na análise do software a ser implementado. As linguagens orientadas a objetos acabam exercendo um papel importante no ensino das idéias de classes e herança ao fornecerem mecanismos para uma efetiva utilização desses conceitos.

Apêndice A

A especificação de algumas classes

As definições das classes a seguir não englobam todas as operações disponíveis na interface. As palavras nos (poucos) comentários não são acentuados em virtude da utilização do ambiente *verbatim* do $\text{\LaTeX}$.

```
//2*****  
//2  
//2    Arquivo reprecjto.hpp  
//2  
//2    Este arquivo contem as definicoes das classes que lidam  
//2    com representacao de conjuntos  
//2  
//2*****  
#include "c:\dilma\proj\include\def.hpp"  
#include "c:\dilma\proj\include\objeto.hpp"  
typedef enum {AVL, BST, DOIS_3 } tp_rpr_cj;  
typedef enum {HASH, LISTA_LIG, LISTA_DUP } tp_rpr_sem;  
//2*****  
//2  
//2          CLASSE    REPR_CONJ.HPP  
//2  
//2    Classe abstrata que engloba/denotas as possiveis  
//2    representacoes para conjuntos.  
//2  
//2*****  
class repr_conj  
{  
char* area;  
public:  
    repr_conj();  
    ~repr_conj();  
    virtual void insere (objeto&);  
    virtual void remove (objeto&);  
    virtual boolean pertence (objeto&);  
protected:  
    virtual objeto& prox_elem (char*);
```



```

};
//2-----
//2
//2          Classe repr_sem
//2
//2
//2
//2-----
class repr_sem : public repr_conj
{
public:
    repr_sem();
    ~repr_sem();
    // redefinicoes de funcoes do pai
    void insere (objeto&);
    void remove (objeto&);
    boolean pertence (objeto&);
    // A funcao abaixo eu queria que so a classe
    // it_repr tivesse acesso
    protected:
        objeto& prox_elem (char*);
};
//2-----
//2
//2          Classe repr_com
//2
//2
//2
//2-----
class repr_com : public repr_conj
{
public:
    repr_com();
    ~repr_com();
    // redefinicoes de funcoes do pai
    void insere (objeto&);
    void remove (objeto&);
    boolean pertence (objeto&);
    objeto& prox_elem (char*);
};
//2*****
//2
//2          CLASSE    IT_REPR.HPP
//2
//2    Classe abstrata que engloba/denota as possiveis
//2    representacoes para conjuntos.
//2
//2*****
class it_repr
{
// variavel para armazenar a informacao de que representacao esta
// sendo percorrida
repr_conj* r;
// variavel para armazenar a indicacao de qual no da representacao
// corrente foi visitado por ultimo

```

```

char* ind;
public:
    it_repr();
    ~it_repr();
    objeto& elemento();
    void inicia_perc();
    boolean fim_perc();
    void anda_perc();
};
//2*****
//2
//2          Arquivo objeto.hpp
//2
//2      Contem a definicao da classe objeto, que quer englobar
//2      todos os elementos do sistema (exceto as representacoes
//2      e iterators, me preocupo apenas em pegar os caras pas-
//2      siveis de armazenamento em conjuntos e afins)
//2
//2*****
#include "c:\dilha\proj\include\def.hpp"
//2*****
//2
//2          Classe      OBJETO.HPP
//2
//2*****
class objeto
{
// trecho que cuida do controle do tipo do objeto
public:
    objeto();
    ~objeto();
    virtual void imprime ();
    friend boolean operator== (objeto&, objeto&);
};
//2-----
//2
//2          Arquivo CONJ.HPP
//2
//2      Contem as definicoes das classes ligadas a conjuntos.
//2
//2-----
#include "c:\dilha\proj\include\def.hpp"
#include "c:\dilha\proj\include\repr_conj.hpp"
#include "c:\dilha\proj\include\objeto.hpp"
//3 como vou usar referencia a classe conj_conj antes de sua defini-
//3 cao preciso fazer um pre-prototipo bem curtinho
class conj_conj;
//2-----
//2
//2          Classe CONJ
//2
//2      Abstracao do tipo conjunto com todas as operacoes
//2      usuais
//2
//2-----
class conj : public objeto
{

```

```

// variavel usada para ligar o objeto conjunto com a sua repre-
// sentacao (estrutura de armazenamento)
repr_sem* r;
// area para a cardinalidade do conjunto
int tam;
public:
    //3 definicao da interface de conj
    conj();
    conj(conj&);
    conj(tp_rpr_sem);
    ~conj();
    //5 protected: As duas funcoes abaixo deveriam ser de uso restrito
    //5             as subclasses de conj, mas nosso compilador ainda
    //5             nao incorpora esse tipo de acesso.
    //5
    //5             Pareceria legal colocar como private, mas nesse caso
    //5             as operacoes das classes derivadas que especializam
    //5             os virtuais nao poderiam usar as duas funcoes abaixo
    void inc_tam();
    void dim_tam();
    virtual void insere (objeto&);
    virtual remove (objeto&);
    virtual boolean pertence (objeto&);
    friend boolean operator== (objeto&, objeto&);
    friend boolean operator<= (objeto&, objeto&);
    friend boolean operator>= (objeto&, objeto&);
    friend boolean operator< (objeto&, objeto&);
    friend boolean operator> (objeto&, objeto&);
    friend boolean operator!= (objeto&, objeto&);
    boolean e_vazio ();
    boolean e_minimal (conj_conj&);
    friend conj& operator= (objeto&, objeto&);
    friend conj& operator+ (objeto&, objeto&);
    friend conj& operator- (objeto&, objeto&);
    friend conj& operator* (objeto&, objeto&);
    virtual imprime();
    int card();
    objeto& um_elemento();
    //3 as funcoes abaixo se relacionam com a troca de representacao
    //3 durante a execucao ou atrelamento de multiplas representacoes
    troca_repr (tp_rpr_cj);
    outra_repr (tp_rpr_cj);
};
//2-----
//2
//2             Classe conj_ord
//2
//2 Classe dos conjuntos com relacao de ordem total entre
//2 os seus elementos. Um conjunto com ord tem a vantagem de
//2 poder ser armazenado em uma arvore 2-3, arvore de busca
//2 binaria e etc. Esta atrelado a classe de representacoes
//2 gerais, porque qualquer representacao de conjunto serve.
//2
//2 E derivado da classe conj. Tem duas operacoes a mais que
//2 a classe base: min e max

```



```

//2
//2-----
class conj_ord : public conj
{
    repr_conj* r;
public:
    conj_ord();
    conj_ord(conj_ord&);
    conj_ord(tp_rpr_cj);
    ~conj_ord();
    //3 Alem dos constructors ganhamos apenas duas operacoes: max e mim.
    objeto& max();
    objeto& min();
    //3 Preciso colocar na interface a lista de operacoes redefinidas
    //3 em relacao ao pai conj
};
//2-----
//2
//2          Classe CONJ_CONJ
//2
//2 Classe de conjuntos com a restricao de que os objetos
//2 armazenados sao conjuntos. Acrescenta-se apenas duas
//2 novas operacoes a interface: uniao/interseccao da
//2 familia.
//2-----
class conj_conj : public conj
{
public:
    // construtores e destrutores
    conj_conj();
    ~conj_conj();
    conj_conj(conj_conj&);
    // Precisa listar as rotinas herdadas que serao redefinidas
    // Operacoes que resultam na interseccao/uniao dos elementos
    // do objeto de tipo conj_conj
    conj& inter_fam();
    conj& uniao_fam();
};
//2-----
//2
//2          Classe it_conj
//2
//2 Classe de iterators para conjuntos.
//2
//2
//2-----
class it_conj
{
    // variavel que guarda qual conjunto esta sendo percorrido pelo
    // objeto iterator em questao
    conj* qual;
    // armazenamento de apontador pra estrutura do atual elemento do
    // percurso
    char* onde;
public:
    it_conj();

```

```

    ~it_conj();
    objeto& elemento();
    void inicia_perc();
    boolean fim_perc();
    void anda_perc();
};
//2-----
//2
//2          Classe matriz
//2
//2 Define uma classe de matrizes de uma, duas ou tres dimensoes com
//2 quaisquer conjuntos como indices (poderiamos definir de qual-
//2 quer dimensao se o usuario desse os respectivos conjuntos
//2 indexadores em um conjunto!!!!)
//2
//2 Numa primeira implementacao e' bom restringir os elementos
//2 armazenados a numeros, a fim de nao complicar a implementacao
//2 de operadores (nao que seja grande a complicacao)
//2-----
#include "c:\dilha\proj\include\objeto.hpp"
#include "c:\dilha\proj\include\repr_tab.hpp"
class matriz : objeto
{
    repr_tab* tab;
    int dim;
public:
    matriz(conj&);
    matriz(conj&,conj&);
    matriz(conj&,conj&, conj&);
    matriz(matriz&);
    associa(conj&, conj&, conj&, int);
    // operadores
    friend float& operator[](matriz&,conj&,conj&,conj&);
};
//2-----
//2
//2          Arquivo Grafo.hpp
//2
//2 Contem as definicoes das classes grafo e grafo_dir
//2
//2
//2-----
#include "c:\dilha\proj\include\def.hpp"
#include "c:\dilha\proj\include\objeto.hpp"
#include "c:\dilha\proj\include\conj.hpp"
#include "c:\dilha\proj\include\repr_gr.hpp"
//2-----
//2
//2          Classe GRAFO.HPP
//2
//2
//2
//2
//2-----
class grafo : public objeto
{

```

```

// variavel para representacao
repr_gr* rg;
// conjunto de vertices
conj* VG;
// conjunto de arestas
conj* AG;
public:
    grafo(conj&, conj&);
    grafo(grafo&);
    grafo(conj&, conj&, tp_rpr_gr);
    friend grafo& operator= (grafo&, grafo&);
    friend boolean operator== (grafo&, grafo&);
    friend boolean operator<= (grafo&, grafo&);
    friend boolean operator>= (grafo&, grafo&);
    grafo& subg_aresta (conj&);
    grafo& subg_vert (conj&);
    conj& conj_arest ();
    conj& conj_vert ();
    void insere_arest (objeto&);
    void insere_arest (objeto&, objeto&);
    void insere_vert (objeto&);
    objeto& ponta1 (objeto&);
    objeto& ponta2 (objeto&);
    grafo& remove_arest(objeto&);
    grafo& remove_vert (objeto&);
    grafo& contrai_arestas(conj&);
    grafo& contrai_verts (conj&);
    // parte do peso e marcacoes
    friend void associa_peso(grafo&, peso&);
    friend void associa_marca(grafo&, peso&);
    // mudanca de representacao e multiplas representacoes
    void trocarepr (tp_rpr_gr);
    void outrarepr (tp_rpr_gr);
    circuito& um_circuito();
    circuito& circ_com_vert (conj&);
    circuito& circ_com_arest(conj&);
    caminho& um_caminho(objeto&, objeto&);
    caminho& cam_com_vert (conj&);
    caminho& cam_com_arest(conj&);
    grafo& uma_componente();
    grafo& uma_componente(objeto&); // dado um vertice ou aresta
    grafo& uma_bi_componente();
    conj& estrela(objeto&);
    conj& delta (conj&);
    boolean e_conexo();
    boolean e_completo();
    boolean e_simples();
    boolean e_bipartido();
    boolean e_arvore();
    grafo_conexo& kn(int);
    grafo_conexo& knm(int,int);
    grafo& operator+(objeto&);
    grafo& operator-(objeto&);

```



```

    conjunto& aresta (u, alfa, v);
};
class graf_dir : public grafo
{
    // varias das operacoes herdadas devem ser redefinidas
public:
    objeto& inic (objeto&);
    objeto& fim  (objeto&);
    void muda_direc(objeto&);
    conj& atingiveis (conj&);
    conj& arest_entram (conj&);
    conj& arest_saem  (conj);
    circ_dir& um_circ_dir();
    circ_dir& circ_dir_com_vert (conj&);
    circ_dir& circ_dir_com_arest(conj&);
};

```

Apêndice B

Alguns exemplos

A seguir ilustramos a utilização das rotinas definidas na biblioteca para especificar alguns algoritmos “abstratos” em grafos, ou seja, algoritmos cuja descrição não envolve estruturas de dados.

1. Feofiloff e Lucchesi definem em [F-L 88] (página 15) o seguinte algoritmo:

Algoritmo de Busca

“O algoritmo é iterativo. Cada iteração começa com uma parte X de V_G . A primeira iteração começa com $X = Q$. Cada iteração consiste no seguinte:

Se existe β em $\vec{b}X$ então comece nova iteração com $X \cup \{v_+\beta\}$ no papel de X .
Se existe γ em $\vec{c}X$ então comece nova iteração com $X \cup \{v_-\gamma\}$ no papel de X .
Senão, devolva X e pare.”

Nosso *programa* para esse algoritmo é:

```
conj& Busca (grafo& G, conj& Q, conj& b, conj&c);
{
    conj X = Q;
    conj bX, cX;
    boolean faz = true;
    while (faz)
    {
        faz = false;
        bX = b * G.arestas_entram(X);
        if (bX.e_vazio() == false)
        {
            X.insere (bX.um_elemento());
            faz = true;
        };
        cX = c * G.arestas_saem(X);
        if ( cX.e_vazio() == false )
        {
            X.insere (cX.um_elemento());
            faz = true;
        }
    }
    return X;
}
```

2. Nosso próximo exemplo é de [F-L 88], página 22:

Algoritmo do Caminho Mínimo

“O algoritmo tem dois estágios, ambos iterativos. O primeiro estágio calcula X e o segundo calcula C . Cada iteração do primeiro estágio começa com uma seqüência $X_0, X_1, \dots, X_m$ de conjuntos-sorvedouro. A primeira iteração começa com $m = 0$ e $X = Q_+$ onde Q_+ é o conjunto dos termos de todos os caminhos diretos que começam em Q . Cada iteração consiste no seguinte:

Caso 1: $X_m \cap P = \emptyset$ e $aX_m = \emptyset$. Neste caso devolva X_m e pare.

Caso 2: $X_m \cap P = \emptyset$ mas $X_m \neq \emptyset$. Seja X_{m+1} a união de X_m com o conjunto das origens de todos os caminhos elementares que terminam em X_m . Comece nova iteração com $X_0, \dots, X_m, X_{m+1}$ no papel de $X_0, \dots, X_m$.

Caso3: $X_m \cap P \neq \emptyset$.

Escolha um vértice x_0 em $X_m \cap P$, devolva $\{X_0, \dots, X_{m-1}\}$, e inicie a execução do segundo estágio com $\langle x+0 \rangle$ no papel de $\langle x_0, \xi_1, \dots, \xi_n, x_n \rangle$ de P a X_k . Cada iteração consiste no seguinte:

Se $k = 0$ então devolva $\langle x_0, \xi_1, \dots, \xi_n, x_n \rangle$ e pare.

Senão, seja $\langle z_0, \zeta_1, \dots, \zeta_j, z_j, \alpha, z_{j+1} \rangle$ um caminho elementar com origem x_n e término em X_{k-1} , e comece nova iteração com a concatenação

$$\langle x_0, \xi_1, \dots, \xi_n, x_n \rangle \cdot \langle z_0, \zeta_1, \dots, \zeta_j, z_j, \alpha, z_{j+1} \rangle$$

no papel de $\langle x_0, \xi_1, \dots, \xi_n, x_n \rangle$ e com $X_0, \dots, X_{k-1}$ no papel de $X_0, \dots, X_k$.”

Para este algoritmo temos o seguinte programa baseado nas rotinas de nossa biblioteca:

```
void Caminho_Minimo (grafo-dir& g, conj& P, conj& Q,
                    conj& X, conj-conj& XX,
                    circuito& C, int qualresp)
{
    conj XM;
    X = g.atingiveis (Q);
    seq_conj SX;
    SX.insere(X);
    boolean aindafaz = true;
    qualresp = 0;
    while (aindafaz)
    {
        XM = SX.ultimo ();
        if ( XM*P == cjvazio ) // testa se vazio
        {
            if ((g.subgr_vert(XM).conj_arest == cjvazio)
            {
                aindafaz = false;
                qualresp = 1; // esta retornando X
                X = XM;
            }
            else {
                conj AUX = g.arestas_entram (XM);
                conj XM1 = g.atingiveis(AUX);
                SX.insere_fim (XM1);
            }
        }
    }
}
```



```

else { // prepara coisas para segundo estagio
    objeto X0 = (XM * P).um_elemento();
    SX.remove_ultimo();
    // coloca os elementos de SX em XX
    for (int i=1; i <= SX.tam; i++)
        XX.insere (SX.posicao(i));
    circuito CC;
    CC.insere(X0);
    seq_conj SK = SX;
    aindafaz = false;
}
}
if (qualresp == 0)
{
    boolean faz = true;
    while (faz)
    {
        if (SK.card == 1)
        {
            faz = false;
            qualresp = 2;
            C = CC;
        }
        else { conj AUX = g.arestas_entram (SK.elemento(k-1));
            perc_conj P(AUX);
            XN = CC.ultimo();
            objeto& zj = P.elemento();
            P.anda();
            while (!P.fim())
            {
                caminho cz;
                boolean achou;
                um_caminho (zj, xn, achou);
                if (achou)
                {
                    CC = CC + CZ;
                    SK.remove_ultimo();
                }
            }
        }
    }
}
}
}

```

Bibliografia

- [A-K 86] H. Ait-Kaci e R. Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance", *Journal of Logic Programming*, **3**:3 (1986), 185-215.
- [A-S 88] K. Alws e I. Glasner-Schapeler, "Experiences with Object Oriented Programming", *SIGSOFT Software Engineering Notes*, **13**:4 (1988).
- [Au 86] A. B. H. Ferreira, *Novo Dicionário da Língua Portuguesa*, Editora Nova Fronteira, Segunda Edição, 1986.
- [Ba 78] J. Backus, "Can Programming Be Liberated from von Neumann Style?", *Communications of the ACM*, **21**:8 (1978).
- [Bar 86] P. Barth, "An Object-Oriented Approach to Graphical Interfaces", *ACM Transactions on Graphics*, **5**:2 (1986), 142-172.
- [Bas 80] V. Basili, *Models and Metrics for Software Management and Engineering*, IEEE Computer Society Press, 1980.
- [Beck89] K. Beck, "A Laboratory for Teaching Object-Oriented Thinking", *OOPSLA '89*, outubro de 1989, 1-6.
- [Beta] B. Kristensen, O. Madsen, B. Møller-Pedersen e K. Nygaard, "The BETA Programming Language", DAIMI PB - 229, Aarhus University, Computer Science Department, novembro de 1987. Também em *Research Directions in Object-Oriented Programming*, editado por B. Shriver e P. Wegner.
- [B-C 89] J. Briot e P. Cointe, "Programming with Explicit Metaclasses in Smalltalk-80", *OOPSLA '89*, outubro de 1989, 419-431.
- [BCW 86] K. B. Bruce, W. College e P. Wegner, "An Algebraic Model of Subtypes in Object-Oriented Languages", *ACM SIGPLAN Notices*, **21**:10 (1986), 163-172.
- [Bl 82] B. I. Blum, "The Life Cycle – A debate over alternate models", *ACM SIGSOFT Software Engineering Notes*, **7**:4 (1982).
- [Bo 86] G. Booch, "Object-oriented Development", *IEEE Transactions on Software Engineering*, **SE-12**:2 (1986), 211-221.
- [Bob 86] D. Bobrow e outros, "CommonLoops: Merging Common Lisp an Object-Oriented Programming", *ACM SIGPLAN Notices*, **21**:10 (1986).

- [Bob 88] D. Bobrow, "Common Lisp Object", *ACM SIGPLAN Notices*, edição especial de setembro de 1988.
- [Ca 86] T. A. Cargill, "Pi: A Case Study in Object-Oriented Programming", *OOPSLA '86 Proceedings*, setembro de 1986, 350-360.
- [Car 84] "The Semantics of Multiple Inheritance", Proc Colloquium on the Semantics of Data Types, Sophia-Antipolis, 1984.
- [CCHO89] P. Canning, W. Cook, W. Hill e W. Olthoff, "Interfaces for Strongly-Typed Object-Oriented Programming", *OOPSLA '89 Proceedings*, outubro de 1989, 457-466.
- [CGF 86] M. A. Casanova, F. A. C. Giorno e A. L. Furtado, *Programação em Lógica*, V Escola de Computação, Belo Horizonte, 1986.
- [Cook89] W. Cook e J. Palsberg, "A Denotational Semantics of Inheritance and its Correctness", *OOPSLA '89 Proceedings*, outubro de 1989, 433-443.
- [Cox 84] B. J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", *IEEE Software*, 1 :1 (1984), 50-69.
- [Cox 86] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading(Mass.), 1986.
- [CW 85] L. Cardelli e P. Wegner, "On understanding types, data abstraction and polymorphism", *ACM Computing Surveys* 17:4 (1985).
- [C-L 85] B. Cox e L. Ledbetter, "Software-IC's", *BYTE*, junho de 1985.
- [C-O 88] S. Clerici e F. Orejas, "GSBL: An Algebraic Specification Language Based on Inheritance", em [LNCS2].
- [Dew 87] S. C. Dewhurst, "Flexible Symbol Table Structures for Compiling C++ ", *Software — Practice and Experience*, 17:8 (1987), 503-512.
- [Dijk] E. W. Dijkstra, "My Hopes of Computing Science" em [FL 80].
- [DMSV89] R. Dixon, T. McKee, P. Schweizer e M. Vaughan, "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance", *OOPSLA '89 Proceedings*, outubro de 1989, 211-214.
- [Duf 86] C. B. Duff, "Designing an Efficient Language", *Byte*, agosto de 1986, 211-224.
- [D-H 87] "On Some Algorithms for Multiple Inheritance in Object-Oriented Programming", em [LNCS1].
- [D-N 66] O. Dahl e K. Nygaard, "SIMULA - An ALGOL-based Simulation Language", *Communications of the ACM* 9:9 (1966), 671-678.
- [D-T 88] S. Danforth e C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Computing Surveys* 20:1 (1988), pag 29-72.
- [FDFP69] D. Friedman, D. Dickson, J. Fraser e T. Pratt, *Graspe 1.5 - A Graph Processor and Its Application*, Universidade de Houston, 1969.

- [FL 80] H. Freeman e P. Lewis II (editores), *Software Engineering*, Academic Press Inc., 1980.
- [Fr 80] P. Freeman, "The Central Role of Design in Software Engineering: Implications for Research" *em* [FL 80].
- [F-L 88] P. Feofiloff e C. Lucchesi, *Algoritmos para Igualdades Minimax em Grafos*, VI Escola de Computação, Campinas, SP, 1988.
- [Gl 82] G. R. Gladden, "Stop the Life Cycle, I Want to Get off", *ACM SIGSOFT Software Engineering Notes*, 7:2 (1982).
- [GNV 88] E. Gansner, S. North e K. Vo, "DAG — A program that Draws Directed Graphs", *Software — Practice and Experience*, 18:11 (1988), 1047-1062.
- [Go 84] J. A. Goguen, "Parameterized Programming", *IEEE Transactions on Software Engineering*, SE-10:5 (1984), pag. 528-543.
- [Go 86] J. A. Goguen, "Extensions and Foundations of Object-Oriented Programming", *ACM SIGPLAN Notices*, 21:10 1986, 153-162.
- [Gor 87] K. E. Gorlen, "An Object-Oriented Class Library for C++ ", *Software — Practice and Experience*, 17:2 (1987), 899-922.
- [Gut 89] S. Guthery, "Are the Emperor's New Clothes Object Oriented?", *Dr. Dobbs's Journal*, dezembro de 1989, 80-86.
- [G-M 88] M. C. Gaudel e Th. Moineau, "A Theory of Software Reusability".
- [gnu] *User's Guide to the GNU C++ Class Library*
- [HSS 89] W. Harrison, J. Shilling e P. Sweeney, "Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm", *OOPSLA '89*, outubro de 1989, 85-94.
- [Hul 84] J. Hullot, "Ceyx, Version 15: I — une Initiation", Rapport Technique no. 44, INRIA, Rocquencourt, Eté 1984.
- [Hum 88] W. Humphrey, "The Software Engineering Process. Definition and Scope", *em* Proceedings of the Fourth International Process Workshop, Washington, 1988.
- [Hut 87] N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy e E. Jul, *The Emerald Programming Language Report*, Technical Report 87-10-07, Dept of Computer Science, University of Washington, Seattle, Washington, 1987.
- [Ho 83] E. Horowitz, *Fundamentals of Programming Languages*, Computer Science Press Inc., 1983.
- [Hor 87] C. Horn, "Conformance, Genericity, Inheritance and Enhancement", *em* [LNCS1]
- [HM 84] E. Horowitz e J. B. Munson, "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, 10:5 (1984).

- [H-O 87] D. C. Halbert e P. D. O'Brien, "Using Types and Inheritance in Object-Oriented Languages", *em* [LNCS1].
- [I-B 82] D. H. H. Ingalls e A. H. Borning, "Multiple Inheritance in Smalltalk-80", *Proceedings of the National Conference on Artificial Intelligence*, Estados Unidos, agosto de 1982, 234-237.
- [Isn 82] J. F. Isner, "A Fortran Programming Methodology Based on Data Abstraction", *Communications of the ACM*, **25**:10 (1982), 686-697.
- [Jer 89] M. E. Jerrell, "Function Minimization and Automatic Differentiation using C++ ", *OOPSLA 'S Proceedings*, outubro de 1989, 169-173.
- [Jul 88] E. Jul, *Object Mobility in a Distributed Object-Oriented System*, Dissertação de doutoramento, University of Washington, 1988.
- [J-K 87] J. P. Jacky e I. J. Kalet, "An Object-Oriented Programming Discipline for Standard Pascal", *Communications of the ACM*, **30**:9, (1987), 772-781.
- [Ke 86] R. Kerr, "Object-Based Programming: A Foundation for Reliable Software" *em* *Proceedings of SIMULA User's Conference*, Stockholm, 1986. Um resumo deste artigo pode ser encontrado em [Ke 87]
- [Ke 87] R. Kerr, "A Materialistic View of the Software Engineering Analogy", *ACM SIGPLAN Notices* **22**:3 (1987).
- [KMPN87] B. Kristensen, O. Madsen, B. Møller-Pedersen e K. Nygaard, "Classification of Actions or Inheritance also for Methods", DAIMI PB - 231, Aarhus University, Computer Science Department, agosto de 1987. Também em [LNCS1].
- [KMPN89] B. Kristensen, O. Madsen, B. Møller-Pedersen e K. Nygaard, "Coroutine Sequencing in BETA", DAIMI PB - 235, Aarhus University, Computer Science Department, janeiro de 1989, segunda edição.
DAIMI PB - 235, Aarhus University, Computer Science Department, janeiro de 1989, segunda edição.
- [K-M 88] J. L. Knudsen e O. L. Madsen, "Teaching Object-Oriented Programming is More Than Teaching Object-Oriented Programming Languages" *em* [LNCS2]
- [K-P 86] T. Kaehler e D. Patterson, *A Taste of Smalltalk*, W. W. Norton Co., New York, 1986.
- [La 81] S. Mac Lane, "Mathematical Models", *The American Mathematical Monthly* **88**:7 (1981), 462-472.
- [Lad 88] R. M. Ladden, "A Survey of Issues To Be Considered in the Development of an Object-Oriented Development Methodology for Ada", *ACM SIGSOFT Software Engineering Notes* **13**:3 (1988), 24-31.
- [LHR 88] K. Lieberherr, I. Holland e A. Riel, "Object-Oriented Programming: An Objective Sense of Style", *OOPSLA '88 Proceedings*, setembro de 1988, 323-334.

- ✓ [L-H 89] K. Lieberherr e I. Holland, "Assuring Good Style for Object-Oriented Programs", *IEEE Software*, setembro de 1989, 38-48.
- [L-R 89] K. Lieberherr e A. Riel, "Contributions to Teaching Object-Oriented Design and Programming", *OOPSLA '89*, outubro de 1989, 11-18.
- [Lis 77] B. Liskov, A. Snyder, R. Atkinson e C. Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, 20:8 (1977), 564-576.
- [LNCS1] J. Bézivin, J.-M. Hullot, P. Cointe e H. Lieberman (Eds.), *ECOOP'87 European Conference on Object-Oriented Programming*, Paris, Proceedings em Lecture Notes in Computer Science 276.
- [LNCS2] S. Gjessing e K. Nygaard (Eds.) *ECOOP'88 European Conference on Object-Oriented Programming*, Oslo, Proceedings em Lecture Notes in Computer Science 322.
- [LNCS3] K. Dittrich (Ed.) *Advances in Object-Oriented Database Systems*, Bad Münster am Stein-Edernburg, setembro de 1988, Proceedings em Lecture Notes in Computer Science 334.
- [MA 88] S. B. T. Mendes e T. C. Aguiar, *Métodos para Especificação de Sistemas*, III EBAI (Escola Brasileiro Argentina de Informática), Curitiba, 1988.
- [Me 88] S. R. L. Meira, *Introdução à Programação Funcional*, VI Escola de Computação, Campinas, 1988.
- [Mel 89] F. Mellender, "An Integration of Logic and Object-Oriented Programming", *ACM SIGPLAN Notices*, 23:10 (1988), 181-185.
- [Mey87a] B. Meyer, "Reusability: The case for object-oriented programming", *IEEE Software*, Março de 1987.
- [Mey87b] B. Meyer, "Harnessing Multiple Inheritance". *Journal of Object-Oriented Programming*, 1:4 (1988), 48-51.
- [Mey87c] B. Meyer, "EIFFEL: Programming for Reusability and Extendibility". *ACM SIGPLAN Notices*, 22:2 (1988), 48-51.
- [Mey 88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, New York, 1988.
- [MiPl85] J. Mitchell e g. Plotkin, "Abstract Types Have Existential Type", em *Proceedings of 12th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, pag 37-51.
- [M-P 88] O. L. Madsen e B. Møller-Pedersen, "What Object Oriented Programming May Be and What it doesn't have to be" em [LNCS2].
- [M-P 89] O. Madsen e B. Møller-Pedersen, "Virtual Classes – A Powerful Mechanism in Object-Oriented Programming", *OOPSLA '89 Proceedings*, outubro de 1989, 397-406.
- [Moo 86] D. A. Moon, "Object-Oriented Programming with Flavors", *ACM SIGPLAN Notices*, 21:11, 1986.

- [My 79] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.
- [Mos 89] E. Moss e outros, "Inheritance: Can We Have Our Cake and Eat it Too?", *OOPSLA '89*, outubro de 1989, 485-490.
- [Ny 86] K. Nygaard, "Basic Concepts in Object Oriented Programming", *ACM SIGPLAN Notices* 21:16 (1986).
- [N-R 89] J. Nielsen e J. T. Richards, "The Experience of Learning and Using Small-talk", *IEEE Software*, maio de 1989, 73-77.
- [N-S 87] K. Nygaard e P. Sørgaard, "The Perspective Concept in Informatics", em *Computer and Democracy*, G. Bjerknes, P. Ehn e M. Kyng ed., A Scandinavian Challenge, Avebury, 1987.
- [OBJ2] J. A. Goguen e J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Software Specifications", em *Specification of Reliable Software*. New York:IEEE Press, 1979, pp 170-189.
- [O-B 89] A. Ohori e P. Buneman, "Static Type Inference for Parametric Classes", *OOPSLA '89 Proceedings*, outubro de 1989, 445-456.
- [Pou 86] D. Pountain, "Object-oriented Forth", *Byte*, Agosto de 1986, 227-233.
- [Pr 82] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Company, 1982.
- [Ren 82] T. Rentsch, "Object-Oriented Programming" *ACM SIGPLAN Notices* 17:9 (1982).
- [Red 88] U. S. Reddy, "Objects as closures: Abstract semantics of object-oriented languages", em *Proceedings ACM Conference on Lisp and Functional Programming*, 1988, 289-297.
- [Ro 81] D. Robson, "Object-Oriented Software Systems", *BYTE*, agosto 1981, 74-86.
- [Sa 80] J. E. Sammet, "Language Aspects of Software Engineering" em [FL 80].
- [Sak88a] M. Sakkinen, "On The Dark Side of C++ ", em [LNCS2].
- ✓ [Sak88b] M. Sakkinen, "Comments on the "Law of Demeter" and C++ ", *ACM SIGPLAN Notices*, 23:12 (1988), 38-44.
- [SB 84] M. Stefik e D. G. Bobrow, "Object Oriented Programming: Themes and Variations", *The AI Magazine*, 1984.
- [Sch86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian e C. Wilpolt, "An Introduction to Trellis/Owl", *ACM SIGPLAN Notices*, 21 :11 (1986), 9-16.
- [Schem] J. Rees e W. Clinger (editores), "Revised Report on the Algorithm Language Scheme", *ACM SIGPLAN Notices*, 21:12 (1986), 37-39.
- [S-HM 83] V. W. Setzer e I. S. Homem de Melo, *A Construção de um Compilador*, Editora Campus, São Paulo, 1983.

- [Set 84] V. W. Setzer, *Manifesto contra o uso de computadores no ensino de primeiro grau*, Editora Antroposófica, São Paulo, 1984.
- [Sny 86] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", *ACM SIGPLAN Notices, OOPSLA '86 Proceedings*, setembro de 1986.
- [Sny 86] Alan Snyder, "CommonObjects: An Overview", *ACM SIGPLAN Notices*, **21:10** (1986), 19-28.
- [Sm 80] A. Goldberg e D. Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley Publishing Company, Reading, Mass., 1983.
- ✓ [S-S 88] E. Seidewitz e M. Stark, "Towards a General Object-Oriented Software Development Methodology", *ACM SIGSOFT Software Engineering Notes*
- [Str 82] B. Stroustrup, "Classes: An Abstract Data Type Facility for the C Language", **13** (1983), 139-161.
- [Str 83] B. Stroustrup, "Adding Classes to the C Language: An Exercise in Language Evolution", *Software — Practice and Experience*, **13** (1983), 139-161.
- [Str 84] B. Stroustrup, "A Set of C++ Classes for Co-routine Style Programming", Computing Sciences Technical Report 90, AT&T Laboratories, 1984.
- [Str86a] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Mass., 1986.
- [Str86b] B. Stroustrup, "An Overview of C++ ", *ACM SIGPLAN Notices*, **21:10** (1986), 7-18.
- [Str87b] B. Stroustrup, "Multiple Inheritance for C++ ", Proceedings of the Spring'87 EUUG Conference, Helsinque, maio de 1987.
- [Str87c] B. Stroustrup, "The Evolution of C++: 1985 to 1987", Proceedings USENIX C++ Workshop, novembro de 1987.
- [Str87d] B. Stroustrup, "Possible Directions for C++ ", Proceedings USENIX C++ Workshop, novembro de 1987.
- [Str87a] B. Stroustrup, "What is 'Object-Oriented Programming'?", em [LNCS1]. Uma versão mais completa (inclui herança múltipla) apareceu também em *IEEE Software*, Maio de 1988.
- [Ta 88] T. Takahashi, *Introdução à Programação Orientada a Objetos*, III EBAI, Curitiba, 1988.
- [Tes 85] L. Tesler, "Object Pascal Report", *Structured Language World*, **9:3**, 1985.
- [Tri 88] H. Trickey, "C++ versus Lisp: A Case Study", *ACM SIGPLAN Notices*, **23:2** (1988), 9-18.
- [Tu 78] D. Turner, "An Overview of Miranda", *ACM SIGPLAN Notices*, **(21):12** (1986), 158-166.

- [Ung 86] D. M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, Report No. UCB/CSD 86/227 (dissertação de doutoramento), Computer Science Division (EECS), University of California, Berkeley, março de 1986.
- [Ung 87] D. Ungar e D. Patterson, "What Price Smalltalk?", *IEEE Computer*, janeiro de 1987.
- [Ve 86] P. A. S. Veloso, *Tipos (abstratos) de dados: programação, especificação, implementação*, V Escola de Computação, Belo Horizonte, 1986.
- [We 76] J. Weizenbaum, *Computer Power and Human Reason*, W. H. Freeman and Company, San Francisco, 1976.
- [Wei 88] A. Weinand, E. Gamma e R. Murty, "ET++ — An Object-Oriented Application Framework in C++ ", *OOPSLA '88 Proceedings*, setembro de 1988, 46-57.
- [Weg 86] P. Wegner, "Classification in Object-Oriented Systems", *ACM SIGPLAN Notices*, 21:10 (1986), 173-182.
- [Weg 88] P. Wegner e S. B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like", *em* [LNCS2].
- [Wie 87] R. S. Wiener, "Object-Oriented Programming in C++ : A Case Study", *ACM SIGPLAN Notices*, 22:6 (1987), 59-68.
- [Wi1 88] N. Wirth, "From Modula to Oberon", *Software – Practice and Experience*, 18:7 (1988), 661-670.
- [Wi2 88] N. Wirth, "The Programming Language Oberon", *Software – Practice and Experience*, 18 num7 (1988), 672-690.
- [Wolf89] W. Wolf, "A Practical Comparison of Two Object-Oriented Languages", *IEEE Software*, setembro de 1989, 61-68.
- [ZSG 79] M. V. Zelkowitz, A. C. Shaw, e J. D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, 1979.

Índice

A

álgebra, 46
assinatura, 46

B

Beta, 21, 91

C

C++, 30
ciclo de vida, 8
classe, 22, 24, 35
classe abstrata, 107, 116

E

Eiffel, 30, 87
encapsulamento, 35, 54
engenharia de software, 7

F

filosofia de desenvolvimento, 23
friend, 113

G

collection, 84
GSBL, 60

H

herança, 22, 24, 38, 40, 42
herança múltipla, 67, 79
hierarquia, 39, 118

I

instância, 22

L

lista genérica, 107

M

member, 73
member, 113
mensagem, 22, 23, 82
método, 82
modelo físico, 19

O

Oberon, 91
OBJ2, 59
objeto, 22, 23, 35

P

padrão, 20
paradigma, 10
polimorfismo, 64
princípio da substitubilidade, 63
private, 73
programação, 9
programação em lógica, 14
programação funcional, 13
programação por procedimentos, 13
protected, 73
public, 73

R

representações alternativas, 118

S

Simula, 17
Smalltalk, 21, 81
software, 7
Software-ICs, 28
subclasse, 40
subtipo, 40, 61
suporte, 11

T

tipo, 61
tipos, 32, 33
tipos abstratos de dados, 34
tipos existenciais, 48, 49, 52
tipos parametrizados, 112
tipos universais, 66
Trellis-Owl, 88

V

virtual, 73, 76, 77, 109