#pg001#nt080#cm00#

1 Programação e Software

1.1 Ciência da computação

Muitos leitores deste livro podem estar inscritos em seu primeiro curso de ciência da computação. Bem-vindo à ciência da computação! Outros leitores podem estar querendo aprender mais sobre assuntos de desenvolvimento de software orientado por objeto. Bem-vindo a este excitante paradigma! (A palavra paradigma significa “um conjunto estruturas, onde todas contém um elemento em particular.” Random House dictionary)

Tipicamente um primeiro curso de ciência da computação apresenta uma linguagem de programação e enfatiza a programação. Alguns estudantes terminam tal curso com a impressão de que ciência da computação é o estudo de programação. Isso não é verdadeiro.

Software é o produto final de um processo de engenharia que envolve requisitos, especificações, análise e projeto. Software é uma entidade visível e tangível. É um conjunto de instruções que permite um computador digital realizar uma variedade de tarefas. *Software* é um produto freqüentemente embrulhado em embalagem bonitinha. *Software* é um negocio multimilionário.

Uma linguagem de programação proporciona uma notação onde se pode expressar algoritmos e estruturas de informações. O computador pode usar tal notação para realizar tarefas úteis. Mas para muitos cientistas da computação, programas representam o menos produtivo, mais rotineiro e talvez a mais monótona parte do processo de desenvolvimento de software. De fato, alguns cientistas da computação nem mesmo programam.

#pg002#nt070#cm00#

Para outros cientistas da computação, a criação de programas é tudo que a ciência da computação representa. A teoria das linguagens de programação salienta a importância da programação. Mas a ciência da computação é muito mais que programação.

Ciência da computação lida com arte, trabalho criativo e cálculos usados em um computador digital. Ciência da computação é tão teórica quanto prática, tão teórica quanto aplicada. Teoria de autômatos, linguagens naturais e artificiais, aprendizado e indução, informação, estruturas de dados, estudo de complexidade e algoritmos desempenham um papel central e também servem como um reforço teórico para toda a ciência da computação. As maiores áreas de aplicação de ciência da computação incluem sistemas operacionais, *design* de compiladores, estruturas de dados e algoritmos, gráficos, análise numérica, bases de dados, linguagens de programação, inteligência artificial, aprendizado de máquina e engenharia de *software*. Como um estudante de ciência da computação, você estará apto a ter cursos em muitas ou em todas estas áreas.

A maioria das ciências requerem que seus praticantes para expressem suas idéias em uma ou mais linguagens técnicas. Químicos aprendem a linguagem de símbolos químicos, operações e conexões que permitem escrever as equações químicas. Físicos usam a linguagem de cálculos, equação diferencial e outras matemáticas avançadas para expressar seus modelos e idéias. Engenheiros elétricos aprendem a linguagem de diagramas de

circuitos. Cientistas da computação também usam uma variedade de notações e linguagens para expressar seus conceitos e produzir resultados.

Um estudante de física precisa aprender matemática básica para ter uma notação que possa ser usada para discussão e raciocínio sobre física. Um estudante de ciência da computação precisa aprender uma linguagem de programação de alto nível e técnicas de resolução de problemas para ser capaz de raciocinar em cima da computação. Para um cientista da computação, a programação não é nada além do que o cálculo, para um físico

Cientistas da computação, como seus colegas de ciências naturais e engenharia, estão preocupados com a construção de modelos, abstrações, análises, design, e implementação. Um programa ou sistema de *software* freqüentemente representa o passo final de um processo de resolução de problemas.

Este livro introduzirá técnicas de dedução e resolução de problemas usando objetos. O princípio fundamental da programação orientada por objeto será explorado e introduzido. Durante esta exploração muitos princípios importantes de computação serão revelados.

#pg003#nt065#cm00#

1.2 Programas de computador

Um programa consiste em uma seqüência de instruções escritas numa linguagem precisa chamada linguagem de programação. Estas instruções são traduzidas em um compilador, para uma linguagem de baixo nível, a linguagem de máquina, para que o computador possa entender.

As aplicações de um software são definidas geralmente em duas categorias: programas de sistemas e programas de aplicações. Programas de sistemas visam controlar um componente do computador, como um dispositivo de armazenamento, dispositivo de saída, ou o próprio computador (ex.: sistema operacional). Programas de aplicações resolvem um problema externo ao computador como um sistema bancário, sistema de controle de tráfego aéreo, sistema de processamento de textos, planilha eletrônica ou outra área de aplicação.

Programas de computador representam o produto final do processo de desenvolvimento de *software*. Eles são entidades tangíveis que podem ser liberados para o cliente, vendidos, embalados. Programas comerciais usualmente vêm acompanhados de um guia do usuário e outros documentos de apoio escritos.

1.3 Linguagens de programação

Três categorias principais de linguagem de programação têm sido desenvolvidas: linguagem de máquina, linguagens assembly e linguagens de alto nível. Os primeiros computadores somente podiam ser programados usando linguagem de máquina. Tal linguagem usa seqüências de zeros e uns (*bits*) que representam instruções precisas para computação e acessos de dados.

Linguagens assembly usam caracteres alfabéticos para representar as configurações de *bits* da linguagem de máquina. As letras usadas descrevem as operações a serem realizadas. Linguagens assembly representam um nível mais alto de abstração do que as

linguagens de máquina. Algumas linguagens assembly modernas suportam estruturas de controle que antes eram previstas somente em linguagens de alto nível.

Linguagens de alto nível assemelham-se com linguagem natural. Dados e operações são representados por declarações descritivas.

Um exemplo: suponha que desejamos adicionar dois números e depositar a soma em um terceiro número. Na maioria das linguagens de alto nível, estas operações são assim simbolizadas:

```
c := a + b
```

#pg004#nt080#cm00#

Os operandos a e b representam os dois números a serem adicionados e a variável c representa o total. O operador ":=" é o operador de atribuição. Ele significa que a soma dos valores em a e b que será atribuído a c[1].

Numa típica linguagem assembly, as instruções poderiam ser lidas:

```
LOAD A
ADD B
STORE C
```

Numa linguagem de máquina as instruções poderiam ser lidas:

```
00011000 00000101
00100001 00000011
00101101 11100001
```

Como você pode ver, somente computadores iriam querer ler a linguagem de máquina.

1.4 Estruturação e programação orientada por objetos

A palavra paradigma tem se tornado popular no últimos anos. As pessoas ouvem falar a respeito do paradigma orientado por objeto.

No mundo do desenvolvimento de software, as décadas de 70 e 80 foram dominadas pela abordagem de solução de problemas (paradigma) chamada programação estruturada. As linguagens dominantes da época incluíam FORTRAN, COBOL, ALGOL, PASCAL, ADA e C. Consideráveis investimentos foram feitos no desenvolvimento ferramentas de *software* para apoiar a programação e o processo de desenvolvimento de software durante esse período. Os métodos e técnicas para projeto e analise estruturados apresentados por Larry Constantine, Tom DeMarco e Edward Yourdon tem influenciado de forma significativa a maneira em que amplos e complexos sistemas de *software* são construídos. Poderosas e dispendiosas ferramentas de *software* para auxiliar a engenharia de *software*

[1] NT: Em português lê-se: "c recebe a mais b"

apoiada por computador (CASE[2]) têm sido desenvolvidas para apoiar este paradigma da programação estruturada.

A abordagem estruturada para resolução de problemas decompõe um problema em funções. Essa abordagem é chamada decomposição funcional – uma complexa operação é dividida em operações menores. Cada uma dessas operações menores ainda é dividida em menores e menos complexas operações até que cada operação seja tratável em tamanho e complexidade. Um sistema de *software* é visto como um processo de transformação – processamento de dados de entrada (*input*) através de uma série de transformações funcionais para produzir dados de saída (*output*). Os dados servem como uma entrada para uma determinada função ou para um processo inteiro, são "processados" e produzem uma saída útil. #pg005#nt070#cm00#Por várias gerações de programadores, esta tem sido a abordagem mais natural para solução de problemas. E tem se tornado um meio de vida.

Em meados de 1970 uma importante, mas silenciosa pesquisa sobre um paradigma diferente de desenvolvimento de *software* foi feita pela Xerox Corporation no Palo Alto Research Center (PARC). Em 1980 este esforço culminou no lançamento comercial da linguagem de programação orientada por objetos Smalltalk-80. Inspirado nas idéias de Alan Kay e seus sócios, esta linguagem visava proporcionar ao programador, um ambiente de desenvolvimento de *software* altamente individual e robusto, que de muitas maneiras se parece com o ambiente moderno de estações de trabalho. Um mecanismo de interface gráfica extremamente "amigável" foi um subproduto deste trabalho e mais tarde levou ao desenvolvimento do tipo de ambiente gráfico tipicamente encontrado na maioria das estações de trabalho, assim como também em computadores pessoais. Em 1980 um mouse como aparelho de interface, uma série de janelas, menus, botões, caixas de diálogo, etc., os quais tem se tornado agora bastante usuais, eram praticamente desconhecidos, com exceção de alguns poucos laboratórios de pesquisa.

Mas mais significativamente, o Smalltalk exemplificou uma abordagem totalmente nova do desenvolvimento de software e solução de problemas - a abordagem da orientação por objetos. Nesta abordagem, a decomposição de dados, ao invés da decomposição de funções, tornam-se a idéia central. As funções se tornam ligadas a um modelo de dados e servem a este modelo de dados. A solução de problemas passa a ser a descrição e modelagem de como objetos interagem entre si.

Em meados de 1980, emergiram várias novas linguagens orientadas por objetos de importância considerável. Entre elas podemos citar Objective-C, Eiffel, CLOS, e C++. Todas estas linguagens estão ainda sendo utilizadas hoje embora seja claro que C++ tenha se tornado a linguagem orientada por objetos mais amplamente utilizada. Das linguagens mencionadas, somente Smalltalk e Eiffel são linguagens orientadas por objeto "puras". Por puro se entende que estas duas linguagens não são baseadas em algum substrato que não orientado por objetos, como são C++ ou Objective-C. Nestas construções de linguagens baseadas em C, podem coexistir tanto a programação estruturada quanto a programação orientada por objetos. Isto freqüentemente leva a um modo misto de solução de problemas. Somente Smalltalk e Eiffel oferecem ao programador a possibilidade para fazer

[2] NT: Do inglês C*omputer Aided Software Engineering*

programação exclusivamente orientada por objetos. O autor deste livro considera isto uma grande vantagem do uso destas duas linguagens.

O próximo capítulo explica o orientação por objetos em maiores detalhes.

Outros paradigmas de programação tem sido criados além do estruturado e do orientado por objetos. Programação funcional, exemplificada pela linguagem de programação LISP e programação lógica exemplificada pela linguagem de programação PROLOG são dois exemplos. #pg006#nt090#cm00#Programação funcional tem sido largamente usada em aplicativos de inteligência artificial e PROLOG em aplicativos para aprendizagem de máquinas.

1.5 Ferramentas comuns de *software*

Incluídas entre as ferramentas comuns de *software* normalmente disponíveis para programadores estão editores de programa, processadores de texto, compiladores, *linkers*, *debuggers*, *profilers* e *browsers*. Cada um destes será descrito resumidamente.

Um editor de programa é um sistema editor de textos, simplificado, que permite ao programador entrar com o texto de um programa. Alguns editores de programa, denominados os editores sensíveis ao contexto, proporcionam uma estrutura de sintaxe que permite ao programador usar uma palavra chave em uma determinada linguagem de programação e o editor então gera o resto de uma expressão automaticamente. A maioria dos editores de programas possuem operações semelhantes como *search* (localiza a palavra no texto), *search/replace* (procura e altera uma palavra ou expressão), *autoindent* (faz a identação do texto de um programa com um número específico de espaços ou de tabulações baseado no contexto do programa),e *goto* (move o marcador ou cursor para uma linha especifica no texto).

Um processador de texto é um programa que ajuda na geração de documentos. Ele freqüentemente permite a integração de componentes gráficos com componentes de texto. Os processadores de texto modernos são bem poderosos e complexos.

Um compilador é um programa especifico das linguagens de programação, que traduz o texto do programa escrito em linguagem de alto nível em linguagem de maquina. Este é um processo essencial que precisa ser cumprido para o programa poder funcionar.

Um *linker* é um programa que integra várias partes de um programa que foram compiladas para um código executável compondo o aplicativo. Normalmente, compilar e *linkar* são operações executadas de maneira integrada, com a finalidade de traduzir o texto de um programa para um código executável.

Um *debugger* é programa que permite um aplicativo ser executado sob o controle do programador. A execução do programa pode ser interrompida em lugares pré-determinados do texto de um programa ou pode ser executada passo a passo. O programador pode então, inspecionar valores de várias entidades no aplicativo para determinar se programa está executando suas funções corretamente. Geralmente os *debuggers* são usados quando um problema é detectado no programa.

#pg007#nt090#cm00#

Um *profiler* é um programa executado em conjunto com um programa em desenvolvimento. Ele calcula e informa o tempo que o programa gasta em diversas partes e seções, permitindo ao programador determinar o local que necessita de mais eficiência para modificá-la na versão final. Essencialmente, um *profiler* é um instrumento de análise de desempenho da execução de um programa.

Um *browser* é um programa que permite uma inspeção visual de um programa amplo e complexo. Desta maneira, é um instrumento essencial para grande projetos de *software*, onde um programador precisa ser capaz de inspecionar outra parte de um software, possivelmente desenvolvida por outros programadores; ou inspecionar o código de bibliotecas reutilizáveis de *software.*

1.6 Programação

Um programa de computador é um conjunto de instruções escritas de acordo com as regras de sintaxe de alguma linguagem de programação. As instruções são traduzidas por outro programa de computador chamado compilador.

O compilador gera instruções numa linguagem de baixo nível, entendidas pela máquina, que permitem ao seu computador executar as instruções fornecidas no seu programa. Instruções assim são difíceis de se ler, entender e possivelmente ainda mais difíceis de se escrever diretamente. Felizmente, para a maioria dos programadores, isso raramente será exigido, se é que será.

Um programa bem escrito deve:

- Ser claro e de fácil entendimento para outro programador.
- Resolver o problema especificado corretamente.
- Ser fácil de se modificar caso as especificações do problema sejam modificadas.

1.6.1 Linguagens de programação

Foram criadas centenas de linguagens de programação para ajudar na solução de vários tipos de problemas. Essas linguagens têm sido agrupadas em várias categorias baseadas nas suas características e na abordagem que usam para resolução de problemas. Essas categorias são:

- Linguagens *Assembly* - Essas linguagens são criadas para cada processador específico com um conjunto próprio de instruções de baixo nível. Programas nessa linguagem são difíceis de serem escritos porque a possibilidade de se formular abstrações nestas linguagens é muito limitada. Elas são bastante propensas a erros e não são facilmente adaptáveis, caso as especificações do problema sofram mudanças. Entretanto, programas desenvolvidos em linguagem *assembly* são muito rápidos. As linguagens de programação mais primitivas foram as linguagens *assembly*.
- #pg008#nt085#cm00#Linguagens procedurais – Essas linguagens foram as primeiras linguagens de “alto-nível”. A primeira delas, FORTRAN, foi desenvolvida no começo da década de 50. Ela era usada principalmente para computação matemática e científica. A unidade básica de abstração em FORTRAN é a subrotina. Subrotinas são similares aos serviços contidos numa

descrição de classe. Os dados são passados para uma subrotina através de seus parâmetros. Computação é tipicamente realizada nesses dados e uma saída é retornada como resultado. Outras linguagens procedurais populares são C, Pascal, Algol e PL/1.

- Linguagens funcionais – LISP pode ser vista como a avó das linguagens funcionais. A sigla LISP significa processamento de listas (*LISt Processing*). LISP e suas linguagens derivadas são amplamente usadas na área de inteligência artificial, aprendizagem de máquina e ciência cognitiva. Mesmo tendo algumas aplicações comerciais escritas em LISP, muitos ainda consideram essa linguagem como uma ferramenta de pesquisa.
- Linguagens Lógicas – PROLOG e suas variações proporcionam a oportunidade de formular um conjunto de proposições lógicas e ter deduções derivadas pela linguagem. PROLOG, assim como LISP, tem sido usada como uma ferramenta de pesquisa na área de inteligência artificial e aprendizagem de máquina.
- Baseadas em objetos – Modula-2 e Ada são as duas linguagens baseadas em objeto que mais se destacam. Cada uma suporta a noção de tipo abstrato de dados (a ser explicado mais tarde nesse capítulo). Essas são as primeiras linguagens procedurais a fornecer uma separação clara entre um modelo de dados e os serviços em torno desses dados (entre a visão externa dos dados, pelo usuário, e suas representações internas). Nenhuma dessas linguagens suporta herança.
- Linguagens orientadas por objetos – Simula, desenvolvida na Noruega no fim da década de 60 é a primeira linguagem de programação orientada por objeto. Na década de 70 o Centro de Pesquisa da Xerox, em Palo Alto, fez algumas pesquisas inovadoras no modelo de objeto que levou ao desenvolvimento da linguagem de programação Smalltalk. Essa linguagem foi lançada comercialmente em 1980. Foi logo seguida por C++, Objetctive-C, CLOS (*Common Lisp Object System*), assim como Eiffel e uma séria de outras linguagens orientadas por objeto menos conhecidas. As duas linguagens orientadas por objeto mais populares usadas hoje são C++ e Smalltalk. Eiffel está ganhando popularidade rapidamente mas fica atrás dessas duas no momento da escrita deste livro.

1.7 Objetivos desse livro

Este livro visa propiciar ao leitor uma base sólida nos princípios fundamentais de programação (nesse caso programação orientada por objeto) e na resolução de problemas.

#pg009#nt---#cm00#

A perspectiva orientada por objeto vista neste livro representa uma nova evolução numa tendência de enfatizar abstrações na resolução de problemas usando o computador e o uso do tipo abstrato de dados em particular (que será definido e discutido no próximo capítulo). Este livro introduz a modelagem de objetos. O autor deste livro acredita que um aluno iniciante vai se beneficiar altamente ao aprender logo cedo que o processo de desenvolvimento de *software* não começa escrevendo-se o código de um programa. Na verdade, um processo sistemático de análise e planejamento vem primeiro. É importante

que o leitor aprenda que a programação é apenas uma parte do processo intelectual associado à construção de software e à ciência da computação.

Através de uma abordagem orientada por objetos, você leitor será apresentado à noção simples e atrativa de que um sistema de *software* é composto de objetos interagindo de maneira harmoniosa que se comunicam entre si através de mensagens. Essas mensagens são definidas com precisão numa descrição de uma classe.

Como muitos programadores estão descobrindo, a perspectiva orientada por objetos é bastante distinta da abordagem antiga, onde se começa de baixo para cima (aprendendo primeiramente sobre tipos numéricos, variáveis, operações de atribuição, operações de comparação, controle de fluxo e repetição, e muito mais tarde, sobre o conceito de funções).

Apesar da abordagem usada neste livro ser audaciosa, ela não é radical. A noção de função é apresentada desde o início (no capítulo 2) and usada em todo o resto do livro. O princípio do encapsulamento que une um modelo de dados com uma abstração funcional forma a parte principal do capítulo 2. Apesar de não enfocar em detalhes de programação até o capítulo 4, o leitor será apresentado ao processo de resolução de problemas orientado por objetos nos capítulos 2 e 3.

Eiffel foi escolhida para apoiar este esforço por causa de sua sintaxe relativamente simples, sua consistência, e seu suporte rico e direto à programação com objetos. Entre as várias linguagens orientadas por objetos desenvolvidas nos últimos 10 anos, ela é a mais elegante e talvez a mais poderosa.

Para os leitores que não estão usando este livro numa disciplina inicial em ciência da computação, deixem-me dizer porque C++ não foi escolhida para este livro. Apesar de reconhecer a enorme popularidade de C++ e a possibilidade de que o leitor tenha que acabar se acostumando rapidamente com esta linguagem, eu acredito que a complexidade de C++, que é de certa maneira uma sintaxe misteriosa, sua falta de segurança, seu apoio continuado em artefatos de baixo nível com ponteiros e referências, distraem do objetivo de aprender a resolver problemas usando objetos. C++ não encoraja nem desencoraja a solução de problemas de maneira orientada por objetos. #pg010#nt075#cm00#C++ é uma linguagem híbrida que culturalmente é embutida em “ideologia do C”. Apesar desta ideologia ter demonstrado ser extremamente produtiva, ela não leva ao aprendizado de um novo conjunto de mecanismos para solução de problemas.

Na minha opinião, esta é importante primeiro tornar-se eficiente no processo de solução de problemas orientada por objetos, antes de iniciar o desafio de dominar uma linguagem mais complexa. O “++” em “C++” não é um pequeno incremento da linguagem C.

E acredito que vocês acharão, como eu acho, que Eiffel não é somente uma notável linguagem para se aprender princípios básicos de construção de programas orientados por objetos, mas também uma rica e potente linguagem para usar na solução de problemas reais após você ter dominado os princípios da programação orientada por objetos.

É minha intenção que este livro inspire em vocês interesse e entusiasmo na solução orientada por objetos de problemas e proporcione a vocês com uma base sólida em alguns princípios básicos de ciência da computação.

1.8 Exercícios

- Cite várias maneiras nas quais computadores têm influenciado sua vida.(Por favor restrinja-se no uso de palavras tolas).
- Cite algumas profissões que envolvem um computador.
- Explique os principais componentes de um computador.
- Qual a vantagem no uso de uma linguagem de alto nível sobre uma linguagem de máquina?
- Quais são os passos tradicionais no ciclo de vida de um sistema de *software*?
- Por que você está interessado por computadores ou pela ciência da computação? Você pode manter sua resposta para esta questão em um lugar seguro por alguns anos e reler sua resposta em 3 anos.

#pg011#nt085#cm00#

2 Uma Abordagem Orientada Por Objetos para Resolver Problemas

Este capítulo é sobre objetos e classes e como cada um é usado na construção de um programa. Quando você terminar este capítulo, você terá aprendido: (palavras técnicas importantes são mostradas em negrito):

- Um **objeto** é uma instância de uma **classe.**
- Um objeto tem **dados** e **comportamento.**
- Um objeto pode utilizar **comandos** ou **consultas** de outros objetos.
- Um comando permite modificar os dados mantidos pela instância de uma classe (pelo objeto).
- Uma consulta permite verificar o valor dos dados mantidos pela instância de uma classe.
- O **estado** de um objeto pode ser verificado através das consultas ao objeto (ao valor de seus dados).
- Classes podem ser relacionadas entre si de três maneiras diferentes: **herança**, **associação** e **uso**.
- Algumas classes são **abstratas** já outras são **concretas.**
- Objetos são criados dentro de um **programa** e interagem com outros objetos à medida que o programa executa.

#pg012#nt095#cm00#

2.1 Objeto, objetos em todos os lugares

2.1.1 Objetos ordinários

O que é um objeto? Um objeto ordinário é caracterizado tanto por seu comportamento quanto pelo seu estado interno. Para objetos ordinários, há uma linha que separa o interior do objeto do lado de fora deste. Dentre as características que definem objetos ordinários estão incluídas textura, cor, cheiro, som, ou custo.

Uma criança no início de sua vida é capaz de distinguir objetos que estão ao seu redor. Isto inclui seus pais e as pessoas que tomam conta dela e os objetos próximos ou sobre seu berço. Isto quer dizer que os seres humanos são criaturas orientadas por objetos. À medida que nós envelhecemos nós aprendemos também a caracterizar objetos baseando no seu aspecto e comportamento. Nossa orientação por objetos é fortalecida pela a observação do mundo a nossa volta e, mais tarde, por um processo formal de educação. Por exemplo, em Química nós aprendemos, a compreender um átomo em particular baseados em sua classificação numa Tabela Periódica de Elementos. Em Biologia, nós aprendemos, a classificar as várias espécies de organismos vivosbaseados em uma elaborada classificação hierárquica de espécies.

Quando jovens, nós observamos os objetos a nossa volta, nós freqüentemente distinguimos ou classificamos objetos por suas semelhanças e diferenças de aspecto e comportamento. Mas, surpreendentemente, nós rapidamente aprendemos a classificar tipos similares de objetos como um carro independente de seu aspecto, tamanho, cor ou textura precisos. Uma criança pode naturalmente reconhecer um desenho mal feito de um carro no livro, um carrinho de brinquedo do tamanho de uma caixinha de fósforos, e um carro real bem como sendo diferentes exemplos, encarnações, ou “instâncias” da classificação CARRO. Muitas crianças podem distinguir mais tarde um carro de um caminhão baseando-se não apenas em sua forma precisa, tamanho, cor, ou outro dado "significativo" de um caminhão, mas ao invés, baseando-se em algumas das propriedades visuais abstratas de carros e caminhões. Parece uma característica dos seres humanos a habilidade natural de classificar objetos.Nós chamaremos estas classificações de classes.

A palavra **instância** será usada para significar um membro particular de uma classe. Por exemplo, um carro vermelho do tamanho de uma caixa de fósforo que seja o brinquedo favorito de uma criança é uma instância da classe CARRO (serão usadas letras maiúsculas para classes) da mesma maneira que o esboço mal feito de um carro no livro de uma criança ou o automóvel real na garagem são outras instâncias de CARRO.

Muitas crianças desenvolvem um nível alto de entusiasmo para carros de brinquedo, caminhões, aviões, barcos, e trens. Esta fascinação parece estar baseada na habilidade comum de todos estes objetos moverem-se de um local a outro e, a mais importante, a habilidade que a criança tem de controlar este movimento. Cedo, as crianças são hábeis em obter uma entendimento abstrato de um VEÍCULO. #pg013#nt050#cm00#Esta abstração representa os traços comuns a todos os veículos incluindo carros, caminhões, aviões, barcos e trens (e outros tipos de veículos conhecidos mais tarde, no momento em que sua experiência de vida e poder aquisitivo aumentam). Uma criança parece apta para fazer esta generalização antes de aprender a palavra “veículo”. Como humanos parecemos ser capazes de desempenhar essa tais generalizações de maneira natural. Esta abordagem de classificação serve como base para uma abordagem “orienta por objetos” para solução de problemas.

A classe VEÍCULO, seja formalmente definida ou entendida informalmente, é considerada como sendo uma **classe abstrata** em contraste com uma **classe concreta**, tal como CARRO, CAMINHÃO, AVIÃO, BARCO, E TREM. Raramente uma criança pede para seu pai: “Por favor, traga-me o veículo vermelho do meu quarto”. Uma classe abstrata (como veículo) é aquela que não possui instâncias de verdade, mas pode ser usada para

produzir instâncias de classes concretas (como CARRO, CAMINHÃO, AVIÃO, BARCO, ou TREM). Ela é uma generalização de uma classe concreta.

A abstração da classe VEÍCULO contêm características comuns de CARRO, CAMINHÃO, AVIÃO, BARCO e TREM, formando a base para a noção de **herança**. A classe concreta adquire (herda) características de sua classe pai (a classe abstrata). Cada característica herdada da classe abstrata é achada em instâncias na classe concreta.

Em nossa percepção dos objetos ordinários, é difícil ou quase impossível, definir precisamente as características de cada classe concreta. Nós podemos rapidamente desenvolver a habilidade de identificar um carro com exatidão. Nós podemos nunca desenvolver a habilidade de definir a classe CARRO com precisão. Nossas habilidades de reconhecimento de padrões são baseadas em fenômenos complexos que não podem ser modelados facilmente. Quando nós "modelamos" uma classe CARRO, nós tentamos extrair características essenciais, ignorando os detalhes não essenciais. Entretanto, é confortante saber que embora não possamos modelar um carro com precisão, podemos ao menos guiá-lo.

2.1.2 Objetos como abstração

Quando começamos a pensar, escrever ou falar sobre carros, nós desenvolvemos uma abstração desta entidade. Esta não é uma atividade que vem facilmente ou cedo na vida. Tanto artistas quanto engenheiros precisam desenvolver tal abstração quando tentam representar um carro. Cada um fará isto de maneira diferente. O artista irá enfatizar o contorno, a textura e a cor ao passo que o engenheiro irá enfatizar o formato e o comportamento do carro. O engenheiro, em particular, se preocupa com a relação entre formato e comportamento. O modelo abstrato de um carro desenvolvido por um engenheiro deve unificar formato e comportamento. Esta unificação é chamada de **encapsulamento.**

#pg014#nt095#cm00#

Os aspectos referentes à forma, textura e cor devem ser considerados como os "dados" do carro. Além desses "dados", o engenheiro se preocupa com questões como a capacidade de fazer curvas do carro, sua capacidade de frear, sua aceleração, etc. Estes fenômenos envolvem a reação do veículo a vários estímulos (por exemplo, pisar no pedal do acelerador, pisar no pedal do freio, virar o volante, etc.). O comportamento de um carro é fortemente influenciado por esses "dados". Um carro grande e pesado geralmente requer muito mais potência para ter uma determinada capacidade de aceleração que um carro pequeno e leve. Ele é geralmente menos manobrável que um carro leve.

Para entender as características de aceleração de um carro, os únicos "dados" (daqui em diante chamados de **atributos**) que podem ser relevantes são a massa do carro, torque, fricção dos pneus e coeficiente aerodinâmico. Essas variáveis constituem o estado interno ou atributos da classe CARRO. Outros atributos como sua cor, marca ou preço são irrelevantes. Aspectos como a velocidade do carro podem ser calculados através do conhecimento desses atributos.

Se alguém está desenvolvendo uma abstração onde o carro é um produto comercial, então os atributos que nós devemos usar para representar o estado interno do carro incluem o valor de empréstimo, a taxa de juros do empréstimo, o número de meses do empréstimo,

o número de pagamentos já feitos, e o preço de tabela do carro. Por esses atributos, o "comportamento" do carro como produto comercial pode ser totalmente descrito.

A modelagem de objetos é similar à modelagem de quaisquer entidades da ciência. O nível de detalhes definidos no modelo depende das metas do problema. Se é desejado estudar as propriedades termodinâmicas da combustão de 4 tempos, associada a um motor de combustão interna, então um modelo que inclui os mínimos detalhes é apropriado. Isso incluiria informações sobre a geometria de cada cilindro e a geometria de cada pistão.

Portanto, a descrição de um objeto, a abstração do objeto, é baseada no problema onde o objeto existe. Aspectos do objeto que exercem um papel essencial na descoberta de uma solução para o problema dado devem ser representados no modelo de objeto (a classe) e os aspectos que não são essenciais são ignorados. Uma abstração representa uma descrição simplificada da realidade. O *Dicionário Oxford* (1966) sugere o seguinte sobre abstrações: "O princípio de ignorar aqueles aspectos de um assunto que não são relevantes para o problema a fim de se concentrar mais naqueles que são."

#pg015#nt085#cm00#

2.2 O modelo de objetos

Coad e Yourdon [1] definem um objeto como "uma abstração de algo no contexto de um problema, refletindo a capacidade do sistema em manter informações sobre ele ou interagir com ele; um encapsulamento de valores característicos e seus serviços exclusivos".

Visto que o "contexto de um problema" pode se referir a quase qualquer coisa, os conceitos chave na definição dada acima, são abstrações, informações, "interação com", valores e serviços exclusivos.

Abstração, como mencionado acima, envolve uma separação das características essenciais das não essenciais. Na definição de abstração, as características essenciais (e consequentemente as características não essências) são relativas ao problema que está sendo resolvido. Isto foi ilustrado na seção 2.1.2 com dois modelos de carro, um modelo físico e um modelo de comércio.

O conceito de **informação** e **atributos** da definição de objetos acima, implica em armazenamento de dados. Cada atributo representa um componente distinto do modelo geral de armazenamento de dados. Isto foi ilustrado na seção 2.1.2 com as características de um carro como massa, aderência dos pneus, torque e coeficiente de aerodinâmica. As consultas na seção externa da classe (a parte da classe que é publicamente disponível) permitem que os valores de alguns atributos sejam obtidos.

O conceito de "interação com" e "serviços exclusivos" na definição acima sugere ação e comportamento. Os serviços associados com um objeto descrevem o que pode ser feito com o objeto ou para o objeto. Este é o **comportamento** do objeto. Os comandos na seção externa da classe detalham precisamente quais serviços são acessíveis para os objetos da classe.

O modelo de objetos envolve dois componentes principais: um modelo de dados e um modelo de comportamento. Estes modelos estão contidos na descrição do objeto na classe. O modelo de dados fornece uma especificação precisa do tipo de informação que é

armazenada em cada objeto enquanto que o modelo de comportamento fornece uma especificação precisa dos serviços que podem ser executados no objeto ou pelo objeto. Somente os serviços descritos no modelo de comportamento podem ser executados pelo objeto. Se funções adicionais forem necessárias elas deverão ser acrescentadas no modelo de comportamento da classe que define o objeto em questão.

2.2.1 Um exemplo de modelo de objeto

Vamos considerar um exemplo simples para ilustrar um modelo de objeto. Suponha que desejamos construir um modelo de objeto para um contador(um objeto que serve para contar coisas). Este contador pode ser usado para registrar o número de vezes que um "evento de contagem" ocorreu. Exemplos de contagem podem incluir a contagem de veículos que chegam numa esquina numa simulação de tráfego ou a contagem do número de aviões que aterrizam numa pista de pouso em um determinado período de tempo.

#pg016#nt080#cm00#

Construiremos uma classe de nome CONTADOR para ilustrar o modelo de objeto.

O estado do objeto CONTADOR, uma instância da classe CONTADOR, é totalmente descrito por uma consulta. O valor da consulta, *resultado*, retém o valor total das vezes que o objeto citado incrementou sua conta. Esta consulta especifica a informação contida no próprio objeto CONTADOR de maneira única e completa.

Os comandos (serviços que podem ser executados pelo ou no objeto) incluem:

- *criar* – construir um novo objeto com valor inicial igual a zero
- *incrementar* – adicionar o valor 1 ao estado atual do contador
- *zerar* – muda o valor atual para zero

A figura 2.1 mostra uma descrição gráfica da classe CONTADOR.

Descrição Gráfica da Classe CONTADOR

Esta notação na qual uma classe é envolvida por uma nuvem pontilhada foi criada por Grady Booch [2] e é chamada de "nuvem de Booch" ou apenas diagrama de classe. O nome da classe fica escrito acima da linha horizontal. Abaixo dela estão os comandos e as consultas. O par de parênteses vazios perto de cada comando indica que estes comandos requerem a entrada de uma informação externa.

2.2.2 A metáfora do nome-verbo e nome- substantivo

Introduziremos uma notação para representar as ações que nós podemos executar em um objeto. Continuaremos com o exemplo da classe CONTADOR, apresentado na última seção. As quatro coisas que nós podemos fazer com este objeto são: criar um, aumentar o seu valor em um, acessar o seu *resultado* atual e retornar o seu valor para 0. Estas são as **responsabilidades** do objeto CONTADOR.

#pg017#nt085#cm00#

Suponha que deixemos a entidade *contador_de_carros* representar um objeto contador que é usado em um programa de simulação de tráfego para manter o controle do

número de veículos que chegam a um posto de pedágio de uma ponte. Os comandos e a consulta que podemos efetuar em tal objeto são:

Comandos:

```
contador_de_carros.criar
contador_de_carros.incrementar
contador_de_carros.zerar
```

Consulta:

```
contador_de_carros.resultado
```

Em cada uma dessas ações, o objeto recebendo a ação está conectado à operação no objeto por um conector ponto (“.”). Seria razoável dizer que a notação acima sugere que estamos realizando ações em um objeto particular para os comandos ou obtendo informação do objeto para a consulta..

Para os comandos, o objeto é um substantivo e a ação é um verbo. Para a consulta tanto a objeto e a consulta são substantivos.

2.2.3 Estado Interno

Suponhamos que haja três objetos CONTADOR no nosso programa simulador de tráfego: *contador_de_carros*, *contador_de_caminhoes*, e *contador_de_onibus*[3]. À medida que veículos chegam ao posto de pedágio da ponte, suponha que a seguinte seqüência de ações acontece:

```
contador_de_carros.zerar
contador_de_caminhoes.zerar
contador_de_onibus.zerar
contador_de_carros.incrementar
contador_de_carros.incrementar
contador_de_onibus.incrementar
contador_de_carros.incrementar
contador_de_caminhoes.incrementar
contador_de_carros.incrementar
contador_de_caminhoes.incrementar
```

As três primeiras ações iniciam os três contadores em 0. Depois que as sete ações restantes são efetuadas, os objetos têm o seguinte estado interno: *contador_de_carros (4)*, *contador_de_caminhoes (2)*, *contador_de_onibus (3)*. Os estados internos dos objetos são diferentes uns dos outros por causa das diferentes ações *incrementar* efetuadas em cada um.

#pg018#nt095#cm00#

O ponto principal é que, embora existam três objetos CONTADOR distintos, cada um caracterizado pelo mesmo modelo de objeto (descrição de classe dada na Figura 2.1), os estados internos de cada um destes objetos evoluem dinamicamente.

---

[3] NT: Note que os nomes não tem acentos, uma vez que devemos evitar o uso de acentos em identificadores (nomes usados para representar algo) quando estamos escrevendo numa linguagem de programação qualquer.

2.2.4 Cenários de objetos e mensagens

O diagrama de classe da Figura 2.1 representa um modelo estático da classe CONTADOR. O comportamento dinâmico não é mostrado. Um diagrama de cenário de objeto pode ser utilizado para descrever as interações dinâmicas entre os objetos.

Para cada comando dado no modelo de comportamento de uma classe (ex.: *criar*, *incrementar*, *zerar)*, uma mensagem correspondente a este serviço pode ser enviada para uma instância da classe. Estas mensagens tomam a forma descrita na seção 2.2.3, onde dez ações são citadas. Cada uma destas expressões envolvem o envio de uma mensagem para um objeto.

O comportamento dinâmico da classe CONTADOR é mostrado na Figura 2.2. Esta figura inclui objetos de outras classes que utilizam os objetos da classe CONTADOR.

Diagrama de cenário de objeto

Na figura 2.2, quatro objetos são mostrados através das nuvens sólidas de Booch. O objeto *simulacao* é uma instância da classe SIMULACAO (detalhes não mostrados aqui). O objeto *posto_de_pedagio* é uma instância da classe POSTO_DE_PEDAGIO (detalhes não mostrados aqui). Os objetos *contador_de_carros* e *contador_de_caminhoes* são instâncias da classe CONTADOR.

Os números indicam a seqüência das ações. A primeira ação está associada com o objeto *simulacao* enviando a mensagem *carro_chega* para o objeto *posto_de_pedagio*. A segunda ação está associada com o objeto *posto_de_pedagio* enviando a mensagem *incrementar* para o objeto *contador_de_carros.* #pg019#nt050#cm00#A terceira ação é associada com o objeto *simulacao* de enviando a mensagem *caminhao_chega* para o objeto *posto_de_pedagio*. A quarta ação é associada com o objeto *posto_de_pedagio* emitindo a mensagem *incrementar* para o objeto *contador_de_caminhoes*.

Deve-se enfatizar que somente mensagens que correspondem aos comandos disponíveis são permitidas. Por exemplo, pode ser ilegal para enviar para o objeto *contador_de_carros* a mensagem *aumentar_valor_em_tres*. Tal mensagem com intenção de aumentar o valor de *resultado* em três unidades, somente seria permitida se estivesse incluída na descrição estática da classe CONTADOR.

2.2.5 Parâmetros

Suponha que fosse desejado incrementar o *resultado* do objeto CONTADOR em mais do que um (ex.: muitos carros chegam em faixas diferentes na mesma cabinedepedágio ao mesmo tempo). Um comando adicional pode ser adicionado ao modelo comportamental da classe CONTADOR. Nós podemos especificar este comando como *incrementar_em* (quantia : *INTEGER*). A entidade *quantia* : *INTEGER* dentro dos parênteses indica que *quantia* é um parâmetro e *INTEGER* é a sua descrição ou tipo. O parâmetro *quantia* indica qual o valor a ser adicionado à *resultado*. Nós assumiremos que *quantia* deve ser um valor positivo.

Um diagrama de classes modificado é mostrado na figura 2.3 e um diagrama de cenário de objeto que uso o novo serviço *incrementar_em* é mostrado na figura 2.4.

Descrição modificada da classe CONTADOR

#pg020#nt070#cm00#

Diagrama modificado de cenário de objeto

Um comando pode ter um ou mais parâmetros, cada um de um tipo específico. Eles fornecem valores externos de entrada para o comando que ajudam a determinar a ação executada.

Como exemplo de um comando com vários parâmetros, considere uma aplicação que envolve a construção de uma janela no vídeo de um computador (região geométrica definida por bordas em que os textos e gráficos podem ser mostrados). A informação externa requerida para construir tal janela inclui o comprimento e a largura da janela, além de sua coordenada superior esquerda. A especificação de tal comando em uma classe, JANELA, pode ser:

criar (*canto* : *PONTO*; *largura* : *INTEGER*; *altura* : *INTEGER*)

O primeiro parâmetro, *canto*, é do tipo PONTO. O tipo PONTO é uma classe que inclui entre seus serviços *criar* (*x: INTEGER; y: INTEGER*) para criar um objeto ponto de coordenadas *x* e *y*.

As ações a seguir, dadas por duas mensagens, podem ser tomadas para criar um objeto janela, cujas coordenadas do canto são (5,10), cujo comprimento é 100 e cuja altura é 200.

um_ponto.*criar*(5,10)

uma_janela.*criar*(*um_ponto*,100,200)

O primeiro comando cria e inicializa um objeto ponto (*um_ponto*) com coordenada *x* igual a 5 e coordenada *y* igual a 10. Este objeto ponto é usado para criar e inicializar um objeto janela (*uma_janela*) com canto superior esquerdo dado por *um_ponto* e comprimento de 100 e altura 200.

#pg021#nt080#cm00#

Ambas ações dadas pelas duas mensagens acima dependem da informação externa enviada nos dois parâmetros de *criar* da classe PONTO e os três parâmetros de *criar* da classe JANELA.

Como um exemplo final que ilustra a importância de permitir que os comandos definidos em uma classe incluam parâmetros, considere a classe VEÍCULO. Suponha que queremos criar um veículo com determinada cor, peso, custo e potência. O comando *criar* para a classe VEÍCULO pode ser definido como:

criar (*cor : STRING; peso : INTEGER; custo : REAL; potencia : INTEGER*)

O parâmetro *cor* é do tipo STRING. Essa é a classe que representa uma seqüência de caracteres em modelo de dados (uma palavra comum). As características de tal classe serão discutidas mais a frente no livro. Os parâmetros *peso* e *potencia* são do tipo INTEGER enquanto o parâmetro *custo* é do tipo REAL.

Embora não seja aparente, imediatamente ao leitor e de fato não pareça natural, valores numéricos do tipo INTEGER tem comportamento diferente dos valores numéricos do tipo REAL. Um objeto do tipo INTEGER pode ter apenas valores inteiros. Um objeto do tipo REAL pode ter valores fracionários. Quando fazemos contas com valores do tipo INTEGER, uma resposta exata é computada. Por outro lado, quando fazemos contas em valores do tipo REAL, nem sempre é possível chegar a uma resposta exata. Esse erro de arredondamento presente nas unidades de processamento aritmético de computadores digitais é causada pela capacidade finita de armazenamento para cada casa decimal. Uma quantidade como 1/3 (uma fração recorrente que requer um número infinito de casas decimais) só pode ser representada com precisão finita. Isto leva à introdução de um pequeno erro em qualquer computação que envolva essa quantidade decimal.

Como números do tipo INTEGER se comportam de maneira diferente de números do tipo REAL, seus comportamentos são especificados em duas classes diferentes.

2.3 Relações entre objetos

Raramente nos preocupamos com objetos isolados. Ciência em geral e ciência da computação em particular estão se preocupando com modelagem e compreensão de sistemas. Sistemas orientado por objetos envolvem vários objetos de diferentes tipos trabalhando juntos para atingir desejada meta.

É importante que examinemos os tipos de relação que objetos podem ter uns com os outros. Várias relações importantes que objetos podem ter uns com os outros são baseadas nas relações entre suas classes. #pg022#nt095#cm00#Estas incluem **herança** (vista na seção 2.3.1), **associação** (vista na seção 2.3.2) e **relação de uso** (vista na seção 2.3.3).

2.3.1 Herança

A palavra herança sugere a aquisição de características de um ou mais ancestrais. Este é precisamente o sentido no qual nós deveremos usar este termo em associação com a solução de problemas orientada por objetos.

Suponha que queiramos construir uma nova classe, uma subclasse, que represente uma especialização de uma classe existente. Queremos que a subclasse possibilite o aumento de alguns atributos aos dados da classe mãe, assim como adicionar alguns comandos ou consultas. Também queremos que a subclasse compartilhe as características do modelos de dados da classe mãe e que forneça os serviços da classe mãe.

Há um princípio importante de consistência que deve ser satisfeito quando uma classe é uma subclasse de outra. Este princípio tem três partes:

- A subclasse deve ter uma relacionamento lógico com a mãe que possa ser expresso como “a subclasse é um tipo de” da classe mãe.
- Os atributos da classe mãe devem fazer sentido como parte do estado da subclasse.
- Os serviços da classe mãe devem fazer sentido como parte do comportamento da subclasse.

Como um exemplo, considere a ligação entre a classe VEÍCULO e a classe CARRO. Claramente, classe CARRO é uma especialização da classe VEÍCULO. Há muitos tipos de veículos que não são carros, mas não há carros que não sejam veículos.

Suponha que, para os propósitos deste simples exemplo, a classe VEÍCULO tenha os atributos *cor, peso, velocidade_maxima, e preco.* A classe CARRO tem os atributos adicionais *numero_de_cilindros e cavalos_de_forca.*

Agora vamos considerar a classe AVIÃO, uma outra subclasse da classe VEÍCULO. Em acréscimo aos atributos *peso, velocidade_maxima e preco,* herdados da classe VEÍCULO, ela tem o atributo adicional *envergadura_da_asa.*

Considere agora duas subclasses de AVIÃO: AVIÃO_A_JATO e AVIÃO_A_HÉLICE. A classe AVIÄO_A_JATO apresenta o atributo *impulso_maximo* (uma característica de suas turbinas) e a classe AVIÃO_A_HÉLICE apresenta o atributo *volume_deslocado* (o volume de cada cilindro).

Por ser apenas um simples exemplo não serão feitos esforços no sentido de modelar os comandos e consultas de cada uma destas classes. O diagrama de Booch na figura 2.5 mostra a hierarquia de herança para estas classes básicas de veículo. As setas estão direcionadas das subclasses para as classes mãe.

#pg023#nt070#cm00#

relações de herança para classes VEÍCULO.

2.3.1.1 Classificação

Foi mencionado anteriormente que o processo de classificação pode ser usado para administrar complexidade. Sempre que um grupo relacionado mas de algum modo diferente precisa ser modelado, uma análise cuidadosa de suas semelhanças e diferenças pode levar a uma hierarquia de classes. Atributos que são compartilhados por várias devem ser colocados no topo da hierarquia. Serviços que são compartilhados por muitas subclasses devem também ser colocados no topo da hierarquia. A base principal para classificação é usualmente baseada na distribuição e reuso dos atributos (o modelo de dados).

Como exemplo, vamos considerar o mundo de cachorros de raça pura,. O clube KENNEL americano classificou os cachorros em vários subgrupos, de acordo com suas características físicas e comportamentais. A figura 2.6 mostra um pouco das relações hierárquicas das classes de cachorros de raça.

Várias classes são marcadas com adornos triangulares com a letra “A” colocada no centro do triângulo. O símbolo indica que a classe é uma classe **abstrata**. Uma classe abstrata jamais terá instâncias Seu propósito é agrupar atributos comuns assim como os serviços que são necessários nas classes descendentes. Esses atributos não são mostrados na figura 2.6.

#pg024#nt065#cm00#

Classificações de Cães

Um criador de cães é capaz de saber muito sobre uma raça de cães, conhecendo onde ela fica na hierarquia canina. Do mesmo modo, um engenheiro de software é capaz de saber muito a respeito do comportamento esperado de um objeto por conhecer onde está situado na hierarquia da classe em volta dele.

2.3.2 Associação

Os objetos a nossa volta são geralmente compostos de outros objetos. Seu computador é composto de uma unidade central de processamento, memória de acesso aleatório, memória *cache* de alta velocidade e armazenamento secundário. Cada um destes objetos pode ser dividido em pequeno objetos.Se você continuar a subdividir os componentes, no final, você se encontrará no nível molecular ou atômico. Como sempre, o problema sendo resolvido determina o nível apropriado de granularidade na modelagem de algum objeto como uma "associação" de outros objetos.

Vamos rever novamente o modelo de objeto de um carro. Um carro é composto de um motor, uma transmissão , um chassi, um jogo de rodas e pneus, um sistema elétrico, um sistema de suspensão, um sistema de escapamento e componentes do interior. Cada um destes são partes essenciais do carro e podem ser modelados como classes.

#pg025#nt050#cm00#

O objeto deve ser capaz de satisfazer ao relação "tem_um"em respeito a cada uma de suas partes.

A Figura 2.7 Demostra as relações de associação de um carro.

Relações de associação entre classes.

O círculo escuro conectando a classe *Carro* com seus componetes *Motor*, *Transmissão*, *Suspensão*, *Rodas*, *ComponentesInternos*, *Sistema_de_Escapamento*, e *Sistema_Eletrico* indica ao relação "tem um" de associação.

2.3.3 Relação de uso

Muitas vezes uma classe precisa usar os recursos de uma outra classe que pode não estar intimamente relacionada a ela. Por exemplo, uma classe SIMULAÇÃO pode precisar realizar operações matemáticas, incluindo seno, coseno, e raiz quadrada. Suponha que estes serviços são encontrados na classe MATEMÁTICA.Além disso, a classe SIMULAÇÃO precisa desenhar figuras geométricas na tela. Suponha os serviços para fazer isto são encontrados na classe GRÁFICOS.

A figura 2.8 demostra a relação de uso entre a classe SIMULAÇÃO e as classes MATEMÁTICA e GRÁFICOS. Os círculos claros ligados à classe SIMULAÇÃO indicam a relação de uso com as classes MATEMÁTICA e GRÁFICOS.

#pg026#nt095#cm00#

Uma relação de uso entre classes

2.4 Tipos abstratos de dados

Um tipo abstrato de dados (TAD) é um modelo de dados e um conjunto de operações associadas que podem ser efetuadas por ou em um modelo de dados. A classe CONTADOR definida anteriormente é um exemplo perfeito de um tipo abstrato de dados. O modelo de dados consiste em um simples inteiro,*resultado*. As operações que definem o comportamento desse modelo de dados são *criar*, *incrementar* e *incrementar_em*. O tipo abstrato de dados CONTADOR pode ser entendido como um elemento unificado cujo propósito é contar eventos ou coisas. As coisas que se pode fazer para um objeto contador são zerar seu valor, incrementar seu resultado em 1, ou incrementar seu resultado de um valor inteiro arbitrário não negativo. Isso é tudo. Não se pode adicionar, subtrair, multiplicar ou dividir um objeto CONTADOR por outro objeto CONTADOR ou por um inteiro qualquer. Isso é porque o tipo abstrato de dados definiu propriedades singulares que não incluem tal aritmética.

Tipos abstratos de dados proporcionam poderosas abstrações que podem ser usadas como base para solução de problemas. Os detalhes de baixo nível do modelo de dados (os internos do TAD) se tornam sem importância quando decidimos como os TADs interagem entre si. Apenas as propriedades externas do TAD (definidas pelo conjunto de operações) são importantes na determinação de seu uso.

Tipos abstratos de dados são representados por classes numa linguagem orientada por objetos. Nas linguagens procedurais como C e Pascal, não há sintaxe de linguagem direta ou suporte para tipos abstratos de dados. Programação cuidadosa e disciplinada permite ao programador simular TADs.

O termo "omissão de –dados" *(data-hiding)* é usado significando que o modelo de dados (estado interno) de um TAD não pode ser modificado pelo usuário. O estado interno pode ser modificado apenas através do conjunto de operações definidas pelo TDA. Não há segredo associado com o termo "omissão de dados" *(data-hiding)*, apenas proteção.#pg027#nt080#cm00#O estado interno de um objeto (um TAD) é protegido de corrupções negligentes.

Como se pode definir um tipo abstrato de dados para um semáforo?

O modelo de dados da luz deve permitir os estados internos verde, amarelo e vermelho. Esses valores podem ser definidos tendo um tipo COR_DE_LUZ (tal modelo de dados permitiria somente três valores para uma instância de COR_DE_LUZ).

Os comandos que poderiam ser efetuados na luz incluem:

- *muda_cor* (*cor*: COR_DE_LUZ)
- *prossiga*

A operação *muda_cor* permite ao usuário ajustar o estado interno da luz. A operação *prossiga* muda a cor da luz através da seqüência verde, amarelo, vermelho quando aplicada repetidamente. O comportamento do semáforo é completamente especificado pelas operações *muda_cor* e *prossiga*.

2.5 Produtores e consumidores.

Um sistema de software é geralmente composto de vários componentes individuais. Em uma organização orientada por objeto, esses componentes são dados pelas classes. Uma biblioteca de classes contém uma coleção de classes unidas por seu suportes para alguma área de aplicação. Bibliotecas de classes têm sido construídas para apoiar interfaces gráficas de usuário dentro da programação em Windows, banco de dados para armazenamento de informações complexas, matemática computacional para aplicações em engenharia e ciências, estruturas de dados para representação de tipos abstratos de dados, operações de entrada e saída e outras áreas de aplicações.

Um **produtor** é um programador cujo objetivo principal é a construção de bibliotecas de classes para serem usadas por outros programadores ou por outras partes de uma aplicação. Um **consumidor** é um programador que faz uso da biblioteca de classes para aplicações específicas. Muitas vezes um programador faz o papel dos dois, produtor e consumidor, produzindo algumas classes para depois serem usadas por outras ou na própria aplicação, usando classes já existentes.

É geralmente aconselhável usar recursos disponíveis de uma biblioteca existente para desenvolver uma nova aplicação a menos que você deseje desenvolver novas aplicações a partir de seus princípios básicos (por exemplo reinventar a roda). Um programador instruído pode ser capaz de construir aplicações de software através do uso de componentes de software existentes que são conectados junto com uma pequena porção de código novo. Esta seria uma atividade de consumidor.

#pg028#nt070#cm00#

Como exemplo, considere, uma típica aplicação inicial de programação. Nós desejamos escrever um programa que mostra seu nome na tela de seu computador.

Muitas linguagens de programação são apoiadas por uma ou mais bibliotecas para fazer entrada e saída de dados. Entrada de dados é o processo de interação entre o usuário e computador na qual o usuário transfere informação para o programa que você está usando. Saída de dados é o processo de interação entre o usuário e o computador no qual um programa transfere informação ao usuário.

O primeiro programa que nós desejamos construir envolve somente uma saída simples. O programa deverá mostrar o nome do usuário no terminal de vídeo.

A identação segue o estilo geral do Eiffel.

Um primeiro programa: Mostrando seu nome.

```
class PRIMEIRO_PROGRAMA

creation
   inicio

feature
   inicio is
      do
         io.putstring(“Meu nome e’ xxx”)
      end;
```

```
end -- class PRIMEIRO_PROGRAMA
```

Na listagem 2.1, a classe PRIMEIRO_PROGRAMA contém uma rotina, **inicio**, que inicia a aplicação. Um arquivo de configuração chamado arquivo ACE informa ao sistema EIFFEL que PRIMEIRO_PROGRAMA é a classe de aplicação e especifica que inicio é o ponto de entrada para a aplicação. O leitor deve consultar o seu guia do usuário do sistema EIFFEL para detalhes relacionados com a construção de um arquivo ACE.

A rotina inicio contém somente uma única linha executável de código, *io.put_string* ("Meu nome e' xxx") onde o usuário deve substituir o "xxx" com seu nome. O objeto *io* é definido em uma biblioteca padrão de entrada e saída de dados disponível em todos os sitemas Eiffel. Esta biblioteca é um importante componente reusável de software. A rotina *putstring* é uma das muitas rotinas definidas nesta biblioteca. Ela permite ao usuário mostrar uma "string" (de caracteres) no terminal de vídeo. Uma "string" é uma seqüência de caracteres.

#pg029#nt055#cm00#

Os detalhes formais para escrita de programas em Eiffel e o uso das bibliotecas existentes nele estão introduzidos no capítulo 3. Contudo o leitor pode querer digitar o código da listagem 2.1, compilar e executá-lo. É muito emocionante quando o primeiro programa de alguém é completado com sucesso.

O autor da listagem 2.1 agiu como um consumidor. O recurso que ele consumiu, ou utilizou, foi a biblioteca padrão de entrada e saída de dados. Em capítulos posteriores você verá como inspecionar a interface das bibliotecas e como usar seus recursos. Do começo ao fim deste livro serão usadas rotinas importante de bibliotecas importantes do Eiffel na construção de aplicações específicas. Você também deverá verá o processo de criação de rotinas para uso por outras pessoas.

Deve-se tornar um consumidor competente antes de se tornar um produtor competente. Vários capítulos posteriores enfatizam as responsabilidades do produtor.

2.6 Modelagem de objetos

Modelagem de objetos é envolve análise e *design*. Por causa da natureza introdutória deste livro, somente os conceitos fundamentais de *design* e análise de objetos serão explorados nesta sessão.

2.6.1 Análise

Análise de *software*, tanto orientado por objeto ou não, envolve entendimento e modelagem do problema. Os principais elementos do problema são mapeados em componentes de *software*. A arquitetura inicial destes componentes de *software* é construída de maneira precisa, descrevendo as conexões que existem entre várias entidades do problema. No contexto da orientação por objeto essas entidades são objetos, cada um é uma instância de classe particular.

A análises orientada por objetos envolve a descoberta de classes importantes e suas conexões com outras classes importantes. Como discutido anteriormente, cada classe encapsula um modelo de dados e um conjunto de serviços associados. Estes serviços

representam o modelo de comportamento das classes. Muito do trabalho de análise orientada por objeto envolve determinar o modelo de dados e comportamento de cada classe.

A arquitetura estática, desenvolvida no nível de análise, é dada pelas associações entre classes. Essas associações incluem relações de associação (todo / parte essencial) relação de uso e relações de generalização / especialização (herança). Essas relações são discutidas e ilustradas a seguir.

#pg030#nt095#cm00#

2.6.1.1 Relação de associação

Uma relação de associação é uma relação “todo / parte essencial” ou “intrínseca”. O objeto “inteiro” é “composto da parte”. Essa parte deve ser essencial para a integridade do todo.

A *relação de associação* é um tipo natural de associação. Muitos dos objetos ao seu redor são compostos de partes constituintes. Por exemplo, seu monitor é composto por uma capa de plástico, um tubo de vídeo e botões de controle ou teclas. Cada uma dessas partes constituintes são essenciais para o funcionamento do monitor.

O motor em um carro pode ser considerado a ser uma parte essencial do carro. Embora podendo certamente ser argumentado que o motor tem uma identidade própria (até o seu próprio número de série), pode ser produzido em local separado e pode ser colocado dentro e fora de carros, pela maioria das aplicações o carro é associado com um motor particular que não será mudado. Além disso, o funcionamento do carro é totalmente dependente da presença de um motor. A partir desse ponto de vista, a identidade de um carro (o objeto inteiro) não é separado do motor (a parte constituinte e essencial). O sistema de transmissão do carro poderia também ser considerado uma parte essencial do carro. Alguém poderia, portanto, dizer que o carro (o todo) tem uma relação de associação com o motor e a transmissão (as partes). Há, é claro, muitos outros componentes essenciais do carro que não foram citados.

Num anel de diamante, é justo argumentar que o anel (o objeto todo) tem uma relação de associação com suas partes essenciais, um aro de ouro e uma pedra de diamante. Embora cada uma possa ser produzida separadamente, de um ponto de vista de modelo a identidade d aro de ouro e da pedra de diamante não é crítica. O que é crítico é a identidade de todo o objeto, o anel.

A notação de *Booch* para uma relação de associação é mostrada na figura 2.9. A classe com o retângulo escuro é a “parte” e a classe com o círculo escuro é o “todo”.

Relação de associação

#pg031#nt095#cm00#

2.6.1.2 Relação de Uso

Como exemplo de uma relação de uso, tomemos uma sala, algumas mesas e cadeiras. A sala tem uma identidade sem suas mesas e cadeiras. A sala ainda existe mesmo quando está vazia. As mesas e cadeiras podem ser mudadas, rearranjadas, trocadas, ou facilmente removidas.

A notação de Booch para uma relação de uso é mostrada na figura 2.10. Aqui uma sala é mostrada tendo um relacionamento de uso para zero ou mais cadeiras e zero ou mais mesas.

Relação de Uso

2.6.1.3 Relação de Herança

Herança é um contexto de programa orientado por objeto que implica em especialização. Uma classe pai define atributos gerais e comportamento que são compartilhadas pelas suas crianças. Cada classe criança contém atributos ou comportamento (serviços) mais especializados que não estão presentes no pai.

Por exemplo, um carro pode ser considerado outro tipo especial de veículo. Um carro e um avião podem compartilhar certos atributos (cor, peso, preço) mas têm atributos separados (número de pistões para um carro, envergadura de asa para um avião).

A notação de Booch para herança é mostrada na Figura 2.11 usando as classes VEICULO, CARRO e AVIAO. As setas apontam da criança ao pai (na direção da generalização).

Relação de Herança

#pg032#nt095#cm00#

2.6.2 Análise de um elevador

Discutiremos alguns elementos simples da análise orientada por objeto de um elevador em um prédio comercial.

Quais são os aspectos relevantes para o problema? Existe, é claro, um elevador. Existe um conjunto de botões, cada um com um número indicando um andar em particular do prédio comercial. O elevador é pintado com uma certa cor. Mas isto certamente não é relevante para o funcionamento de um elevador e não será incluído no modelo de análise. Finalmente, existe um usuário, um ser humano que entra no elevador e deseja ser transportado para outro andar, tanto superior quanto inferior, no prédio comercial.

Baseado na descrição do problema, existe uma relação todo/parte entre o elevador e seus botões. Cada elevador tem um conjunto de botões que contêm todos os locais onde o elevador pode ir. Estas são as partes essenciais de um elevador.

Na Figura 2.12, um simples diagrama de classe exibindo a classe ELEVADOR e a classe BOTÃO, é mostrado.

Diagrama de classe para *elevador*

Cada nuvem pontilhada representa uma classe. O nome de classe é dado dentro da nuvem. Como indicado anteriormente, a reta com um ponto em seu começo, que une a classe ELEVADOR à classe BOTAO, indica uma relação todo/parte ou uma relação de associação. A classe com o ponto “tem” ou “é composta” da classe que não tem o ponto.

A notação, 1..*n*, no final da linha indica que existe um ou mais botões que são partes do elevador. O quadrado no final desta linha indica que a classe ELEVADOR contém um conjunto de botões (eles não são compartilhados por quaisquer outros objetos).

Quando uma pessoa entra em um elevador, ele ou ela aperta um botão. Esta ação é mostrada no diagrama de cenário de objeto dado na Figura 2.13.

Cada nuvem sólida representa um objeto especifico com um nome dado dentro da nuvem. Os números seguidos de dois pontos representam a seqüência das ações. Os três eventos que são mostrados na Figura 2.13 são: (1) uma pessoa entra no elevador, (2) a pessoa escolhe um dos vários botões, e (3) a pessoa aperta este botão escolhido.

#pg033#nt085#cm00#

Diagrama de cenário de objetos para uma pessoa entrando no elevador.

O primeiro evento, uma pessoa entrando no elevador, é descrito na figura 2.13 pelo objeto *uma_pessoa* mandando a mensagem *entrar* para o objeto *um_elevador*.

O segundo evento na figura 2.13 é exibido pelo objeto *uma_pessoa* mandando a mensagem *selecionar_botão* para uma coleção de objetos *botões*, mostrado por três nuvens. O botão selecionado é mostrado com o nome *botao_selecionado.*

O terceiro evento exibido é o objeto *uma_pessoa* enviando a mensagem *apertar* para o objeto *botao_selecionado.*

2.6.3 Projeto

Se alguém olha para o problema ao fazer uma análise, deve olhar para a solução quando faz o projeto. A solução é um sistema de *software* contendo vários objetos que interagem uns com os outros. Cada objeto é tipicamente definido por uma classe. Durante o projeto, identifica-se precisamente classes adicionais que interagem com classes principais identificadas na análise para completar uma solução para o problema.

A natureza preliminar deste livro o faz impróprio para detalhar a fase de projeto.

#pg034#nt090#cm00#

2.7 Sumário

- Objetos ordinários são caracterizados por seus comportamentos assim como seus atributos.
- Parece que os seres humanos possuem uma habilidade natural de classificar objetos. Nós chamaremos essas classificações de classes.
- A palavra “instância” será usada para significar um objeto cujas propriedades são descritas numa classe.
- A descrição de um objeto, a abstração do objeto, é baseada no domínio do problema no qual o objeto existe.
- Uma abstração representa uma descrição simplificada da realidade.

- As características do objeto que são essenciais para se achar uma solução do problema dado deve ser representada no modelo do objeto (a classe) assim como características que não são essenciais devem ser ignoradas.
- Coad e Yourdon definem um objeto como “uma abstração de algo no domínio de um problema, refletindo a capacidade de um sistema de manter informação sobre ou interagir com ele; um encapsulamento de atributos e seus serviços exclusivos.
- O modelo de dados fornece uma precisa especificação de qual informação é mantida em cada objeto.
- O modelo de comportamento fornece uma precisa especificação dos comandos que podem ser realizados no objeto.
- Um objeto recebendo uma ação é conectado à operação nesse objeto usando um conector ponto (“.”).
- Um diagrama de classe é usado para descrever a arquitetura estática do sistema de software.
- Um diagrama de cenário de objeto é usado para descrever as interações dinâmicas entre objetos.
- Um serviço específico pode ter um ou mais parâmetros, cada um de tipo específico. Isso fornece uma entrada externa ao serviço que ajuda a determinar a ação realizada pelo serviço.
- Ciência em geral e ciência da computação em particular envolve modelagem e compreensão de sistemas.

#pg035#nt065#cm00#

- Sistemas orientados por objetos envolvem muitos objetos de tipos diferentes trabalhando juntos para alcançar, de alguma maneira, um objetivo desejado.
- O processo de classificação pode ser usado para gerenciar complexidade.
- Sempre que um grupo de objetos diferentes, mas relacionados de alguma maneira, precisarem ser modelados, uma cuidadosa consideração de suas semelhanças e diferenças pode levar a uma hierarquia de classes.
- Atributos que são compartilhados por muitas subclasses devem ser postos em classes no topo da hierarquia.
- A base fundamental para classificação é geralmente baseada na distribuição e reunião de atributos (isto é, um modelo de dados)
- A subclasse deve ter um relacionamento lógico com sua classe mãe, que possa ser expressa como “a subclasse é um tipo de” da classe mãe.
- Os atributos da classe mãe devem todos fazer sentido como parte do estado da subclasse.
- Todos serviços da classe mãe devem fazer sentido como parte do comportamento da subclasse.
- Os objetos em nossa volta geralmente são feitos a partir de outros objetos mais simples.
- Um tipo abstrato de dados (TAD) é um modelo de dados e um conjunto associado de operações que podem ser feitas no modelo de dados.
- Os tipos abstratos de dados fornecem poderosas abstrações que podem ser usadas como base para resolver problemas.

- Torna-se sem importância o interior dos TADs quando decidimos como eles vão interagir entre si. Somente as propriedades externas do TAD (definidas por um conjunto de operações) são importantes na determinação de seu uso.
- Os tipos abstratos de dados são representados por classes em linguagens orientadas por objetos.
- O termo "omissão de dados" é usado para dizer que o modelo de dados (estado interno) de um TAD não pode ser acessado diretamente pelo usuário. O estado interno só pode ser modificado através de um conjunto de operações pré-definidas.
- Um produtor é um programador cujo objetivo principal é a construção de uma biblioteca de classes para outros programadores ou outras partes de um aplicativo.

#pg036#nt000#cm00#

#pg037#nt085#cm00#

- Desenhe um diagrama de Booch de uma hierarquia de veículos. A classe raiz nesta hierarquia deverá ser a classe VEICULO. Indique os atributos associados a cada classe na sua hierarquia de veículos. Os dois primeiros elementos do princípio de consistência estão satisfeitos na sua hierarquia? Explique em detalhes.
- Acrescente alguns métodos (comportamento) para cada classe em sua hierarquia veículo do problema anterior. O terceiro elemento do princípio de consistência foi satisfeito para sua hierarquia?
- Desenhe um diagrama de Booch que descreve as classes que modelam uma sala de aula universitária. Inclua os estudantes, professor, monitor, e qualquer outra coisa que você puder pensar. Mostre as relações que estes objetos têm um com o outro em seu diagrama de classe.
- Ilustre o princípio de composição construindo uma classe que é composta de outros objetos. Desenhe um diagrama de Booch de sua classe.
- Descreva os passos fundamentais envolvendo o ato de escrever um cheque pessoal usando um diagrama de cenário de objeto de Booch. Mostre todos os objetos envolvidos em seu cenário e a sucessão de mensagens enviadas a cada objeto. Descreva com suas próprias palavras o significado de seu diagrama.
- Mostre as associações apropriadas entre as classes que descrevem as entidades seguintes: tulipa, rosa, flor, pétala, abelha, vaso de flor.
- Mostre as associações apropriadas entre as classes que descrevem as entidades seguintes: motocicleta, bicicleta, carro, barco, hidroavião, avião, avião a jato, planador e mobilete.
- Mostre as associações apropriadas entre as classes que descrevem as entidades seguintes: biblioteca, livros, fichário, estantes, seções de estantes, e usuários.
- Mostre as associações apropriadas entre as classes que descrevem as entidades seguintes: universidade, salas de aula, estudantes, professores, quadro-negro, escrivaninhas, terminais de computador, cadeiras, cadernos, e cursos. Se você desejar acrescentar algumas entidades adicionais para enriquecer sua modelagem de objetos, sinta-se livre para fazer isto. Indique para toda classe adicional seu propósito e mostre sua associação com as classes especificadas acima.

#pg038#ntnnn#cm00#

#pg039#nt085#cm00#

3 Os Elementos Básicos de Programas de Eiffel

3.1 Programando

Nós começaremos a programar neste capítulo.

Um sistema de software é uma coleção interconectada de unidades algumas vezes chamadas módulos. Cada módulo contém um conjunto logicamente coerente de operações e um modelo de dados subjacente (que está por trás das operações). Em um contexto de orientação por objetos, um módulo é o mesmo que uma classe: uma unificação de um modelo de dados e comportamento consistindo em um conjunto de funções chamadas consultas e comandos que podem acessar e manipular as informações contidas no modelo de dados. A princípio cada módulo representa um pedaço potencialmente reutilizável de código.

Um ponto de vista de "sistemas" é bem diferente do ponto de vista de um "programa" monolítico mais tradicional. Usando a abordagem do "programa", todo o comportamento desejado do software deve ser embutido e considerado para a única entidade (seu programa). Usando a abordagem de "sistemas", cada módulo é responsável por apenas uma pequena, mas bem enfocada porção do comportamento desejado do software.

Linguagens de programação mais modernas, inclusive Eiffel, favorecem uma abordagem de "sistemas" para construção de *software*. Esta é a abordagem que nós tomaremos desde o começo. Usando esta abordagem, nosso primeiro programa de Eiffel que se denomina "Meu primeiro programa" é dado na listagem 3.1.

Para se ajustar com o mais recente padrão de formatação Eiffel, todos os caracteres são escritos em itálico e em adição, palavras reservadas na linguagem são escritas em tipo negrito.#pg040#nt075#cm00# A indentação também segue o padrão de formatação Eiffel.

Primeiro programa em Eiffel

```
class APLICAÇÃO
creation
   comece

feature

   comece is
      do
         io.put_string("meu primeiro programa")
         io.new_line
      end

end  --class APLICAÇÃO
```

Os dois comandos de saída, *put_string* e *new_line* quando enviados para o objeto padrão de entrada e saída (o objeto *io*) fazem com que a *string* "meu primeiro programa", seja exibida na saída padrão (seu monitor de vídeo) com a posição do cursor movida para a próxima linha. Como a saída e entrada de dados são necessárias e são padrão, o objeto *io* não precisa ser declarado explicitamente. Sua existência ocorre por padrão, em todas as aplicações em Eiffell.

Mesmo para ser capaz de escrever este simples programa, o programador deve ter algum conhecimento dos serviços de saída de dados na classe STANDARD_FILES. Um resumo destes serviços pode ser obtido usando uma ferramenta chamada **short**. Esta ferramenta retira todos os detalhes de implementação e revela somente a informação de interface que poderia ser relevante para o uso desta classe. A listagem 3.2 mostra parte destes detalhes de interface usando o resultado da aplicação do **short** na classe. Somente as rotinas de saída de dados contidas nesta classe são mostradas, Elas proporcionam meios para que o programador possa imprimir caracteres, inteiros, frações decimais (números do tipo ponto flutuante), strings e valores booleanos.

Partes da interface da classe STANDARD_FILES.

```
class interface STANDARD_FILES
    feature specification
       -- rotinas de saída


      #pg041#nt090#cm vários erros de formatação#
    put_char (c : CHARACTER)
    --Escreve 'c' no fim da saída default.

    put_string (s : STRING)
    --Escreve 's' no fim da saída default.

    put_real (r : REAL)
    --Escreve 'r' no fim da saída default.

    put_double (d : DOUBLE)
    --Escreve 'd' no fim da saída default.

    put_int (i : INTEGER)
    --Escreve 'i' no fim da saída default.

    put_bool (b : BOOLEAN)
    --Escreve 'b' no fim da saída default.

    new_line
    --Escreve mudança de linha no fim da saída default.
    --Muitas rotinas não exibidas.

end interface – classe STANDARD_FILES
```

A classe APLICAÇÃO na Listagem 3.1 serve para iniciar a execução do software. Ela é a classe raiz da aplicação. Todos os softwares Eiffel devem estar acompanhados de um arquivo **Ace** que especifica onde estão os arquivo das bibliotecas padrão, os arquivos gerados pelo programador que constituem a aplicação dada, o nome da classe raiz, e o nome da rotina de criação dentro da classe raiz. O arquivo Ace que controla a aplicação dada na Listagem 3.1 é mostrado na Listagem 3.3.

Arquivo Ace para Listagem 3.1 (Usando sistema ISE Eiffel 3)

```
system test

root aplication (ROOT_CLUSTER): “comece”

default

     assertion (all);
   precompiled (“$EIFFEL3/precomp/spec/$PLATFORM/base”)

cluster
   ROOT_CLUSTER: “/disk2/EIFFELWORK3/WORK”;

end
```

#pg042#nt085#cm erro no layout#

O Leitor pode querer consultar o manual de instruções que vem junto com o sistema Eiffel para maiores detalhes a respeito de arquivos Ace.

3.2 A linguagem Eiffel

A linguagem nasceu no final dos anos 80. É uma linguagem totalmente orientada por objeto. Isto implica que as funções podem ser chamadas somente através de objetos e não sozinhas, como entidades lógicas auto-suficientes. O que é um contraste evidente para a popular, mas complexa linguagem orientada por objetos C++ que admite uma mistura de programação orientada por objetos e estruturada na mesma aplicação, sendo denominada linguagem híbrida.

Como uma linguagem totalmente orientada por objeto, a classe em Eiffel é a unidade básica lógica do encapsulamento bem como a unidade básica física, um módulo. Um programa em Eiffel é organizado como um conjunto de classes interligadas e cooperando entre si.

Nas diversas seções seguintes os elementos básicos da programação em Eiffel serão apresentados. Muitos desses elementos são similares aos de outras linguagens. O projetista do Eiffel, Bertrand Meyer, tem sido um grande estudioso da tecnologia de linguagens de programação e projetou uma sintaxe que utiliza algumas das melhores características

encontradas nas outras linguagens. Acredito que você irá achar a sintaxe do Eiffel coerente, lógica e legível.

3.3 Criando e destruindo objetos

Programas são constituídos de classe que criam objetos. Estes objetos são criados, realizam suas tarefas, e são normalmente destruídos. Quando um objeto é criado, um espaço na memória é reservado para seu armazenamento. Quando o mesmo é destruído o espaço reservado é liberado e poderá ser reutilizado por outros objetos criados depois.

Um objeto em Eiffel, e nas outras linguagens orientadas por objetos, é uma instância de uma classe. Seus atributos são dados por um modelo de dados de sua classe. Os comandos que ele pode responder (as rotinas que se pode solicitar através dele) são especificados por um conjunto de rotinas dadas na descrição da classe.

Antes de um objeto ser criado no Eiffel, ele deve ser declarado para se tornar uma variável de um certo tipo. Este tipo é o nome da classe da qual o objeto será uma instância (depois de criado). Quando uma declaração de tipo em Eiffel como, *meu_carro* : CARRO, é dada, o compilador verifica se a classe referida foi definida. Normalmente o arquivo que define a classe deve estar no mesmo subdiretório da classe na qual a declaração existe.

#pg043#nt070#cm vários erros de identação na listagem#

A declaração *meu_carro*: CARRO não cria uma instância da classe CARRO. Nenhuma memória é reservada para o objeto *meu_carro*, como em qualquer outro objeto que foi apenas declarado, assumindo o valor Void(que significa vazio). Neste estado, o objeto não pode receber qualquer comando ou efetuar nenhuma tarefa proveitosa.

Normalmente, para *meu_carro* se tornar uma instância da classe CARRO e ter um região de memória corretamente alocada para si, uma rotina de criação ou um operador de criação deve ser usado.

Considere o caso onde a classe CARRO não tem nenhuma rotina de criação especificada. O que poderia fazer o objeto *meu_carro* existir (ou seja, reservar um local de armazenamento para ele) poderia ser usar o seguinte operador de criação: *!!meu_carro*. O resultado desta expressão é a criação de um local para armazenar *meu_carro*, mas sem providenciar qualquer inicialização dos atributos que podem ser definidos na classe CARRO. Desta maneira, alguns atributos, como aqueles declarados como sendo do tipo INTEGER, REAL, BOOLEAN ou CHARACTER, assumem inicialmente valores padrão (nulos).

Considere outro caso onde a classe CARRO providencie três rotinas de criação: *criar*, *montar* e *construir*. Uma parte da classe CARRO é a seguinte:

```
class CARRO

   creation
      criar, montar, construir

   feature
      criar(cor: STRING; preço: REAL; peso: INTEGER) is
      -- Detalhes não mostrados
```

```
        end -- criar

        montar(cor: STRING; preço: REAL; peso: INTEGER;
                  potencia: INTEGER) is
        -- Detalhes não mostrados
        end -- montar

        construir(cor: STRING) is
        -- Detalhes não mostrados
        end -- construir
```

Algumas expressões que poderiam criar o objeto *meu_carro* incluem:

```
!!meu_carro.criar(“Branco”, 25600, 3100)
!!meu_carro.montar(“Vermelho”, 12000, 3500, 125)
!!meu_carro.construir(“Azul”)
```

#pg044#nt095#cm00#

Em todas as três expressões acima, o objeto *meu_carro* é criado e inicializado com os valores dados como parâmetros nas várias rotinas de criação. No primeiro caso dado acima (*!!meu_carro*) *meu_carro* é criado mas os atributos assumem seus valores padrão.

Deveria ficar claro nessa discussão que objetos devem ser explicitamente criados ou usando o operador de criação (!!) na frente do objeto que você deseja criar ou usando uma rotina de criação específica em conjunto com o operador de criação, como mostrado nos três exemplos acima.

Como os objetos de Eiffel são destruídos? Os sistemas Eiffel proporcionam “coleta automática de lixo” (garbage collection). À medida que um aplicativo Eiffel roda, um processo de coleta de lixo está rodando em segundo plano e detectando quando uma região de memória não está mais ligada a um nome de variável. Em um momento oportuno o processo de coleta de lixo recicla (efetivamente destrói) o armazenamento desnecessário. O seguinte segmento de código mostra um exemplo de um armazenamento que não é mais necessário:

```
meu_carro : CARRO
!!meu_carro.construir(“Azul”)
!!meu_carro.construir(“Vermelho”)
```

A sentença, *!!meu_carro.construir(“Azul”),* faz com que o armazenamento de memória seja alocado e o nome do objeto *meu_carro* anexado a esse armazenamento. A terceira sentença, *!!meu_carro.construir(“Vermelho”),* faz com que novo armazenamento de memória seja alocado e o nome do objeto *meu_carro* anexado a esse novo armazenamento, deixando o antigo desanexado de qualquer outro nome de objeto.

A seqüência de ações é mostrada na Figura 3.1.

#pg045#nt090#cm00#

Produção de Lixo.

A seção de memória destacada que contém “Azul” pode ser recuperada pelo coletor automático de lixo (*garbage collector*) enquanto o programa está rodando. Não é da responsabilidade do programador de *Eiffel* fazer isto.

3.4 Tipos básicos, valores *default* e atribuições

Existem vários tipos básicos de objetos que não exigem uma criação definida para serem usados. Os importantes são: INTEGER, CHARACTER, REAL e BOOLEAN. Considere as seguintes declarações:

```
um_inteiro      : INTEGER
um_caracter  : CHARACTER
um_real      : REAL
um_boolean      : BOOLEAN
```

Cada um desses objetos de tipos básicos recebe um valor *default* que não é *Void.*

Objetos do tipo INTEGER têm um valor *default* de 0. Objetos do tipo CHARACTER têm um valor *default* do caracter NULL (valor ASCII 0). Objetos do tipo REAL têm um valor *default* de 0.0. Finalmente, objetos do tipo BOOLEAN têm um valor *default* de *false*.

#pg046#nt030#cm00#

Os tipos básicos apresentados na seção anterior têm significado de valores. Isto implica que a declaração de tais objetos causam alocação automatica de memória e atribuição de valores *default* para eles.

Quando um tipo básico de objeto (objeto de origem), é atribuído para outro tipo básico de objeto (objeto de destino), este reside numa região de memória diferente de onde está o objeto de origem. O operador de atribuição do Eiffel é “:=”.

O comando a := b deve ser lido como “a recebe b”.

A figura 3.2 mostra a atribuição entre dois objetos do tipo inteiro.

Atribuição de objetos de tipos básicos

3.5 Tipos de objetos ordinários ou de referência.

Objetos ordinários (objetos não básicos) tem semântica de referência. Isso implica que o programador é responsável pela alocação de memória através de um operador de criação, possivelmente junto com uma rotina de criação como foi discutido na seção 3.3. O valor *default* de um objeto ordinário é *Void*. Após o programador alocar um espaço na memória para um objeto, o nome do objeto é anexado àquela região de memória (ver figura 3.1)

O que significa atribuir um objeto ordinário a outro?

Considere o segmento de código abaixo:

#pg047#nt050#cm00#

```
meu_carro: CARRO
```

```
seu_carro: CARRO
!!meu_carro.criar (“verde”,1000,2000)
seu_carro:=meu_carro
```

A figura 3.3 representa a semântica do segmento de código acima.

Atribuição de objetos de tipos de referência

Depois da atribuição de *meu_carro* para *seu_carro*, ambos nomes dos objetos, são ligados à mesma região de memória. Isto implica que se um dos atributos de *meu_carro* for modificado, através do envio de um comando tal, como *meu_carro.mudar_peso*(2500), o atributo peso de *seu_carro* também será modificado para 2500. Não há dois objetos independentes, mas dois nomes diferentes para o mesmo objeto na memória (só um objeto existe na memória do computador).

3.6 Copiando Objetos

Supondo que nos desejássemos que o objeto *seu_carro* da seção anterior tivesse os mesmos valores de atributos de *meu_carro*, sendo um objeto independente que não é ligado à mesma região de memória de *meu_carro*. Suponha que *meu_carro* tivesse sido criado e inicializado. Isto pode ser feito assim: (1) Crie o objeto *seu_carro* e (2) Use a rotina *copy* que é disponível para todos objetos em Eiffel.

```
seu_carro.copy(meu_carro) --assume que seu_carro foi criado
```

#pg048#nt050#cm00#

É essencial que o objeto *seu_carro* já tenha sido associado à uma região de memória para a rotina *copy* trabalhar. Um erro de tempo de execução será criada e um erro será relatado se você invocar a rotina *copy* sobre um objeto vazio (com o valor *Void*).

A semântica da rotina *copy* é mostrada na figura 3.4.

Figura 3.4. Semântica da cópia.

3.7 Clonagem

Suponha que você deseja alocar memória para *seu_carro* e ao mesmo tempo você pretende copiar os valores dos atributos de *meu_carro* para *seu_carro*. Isto pode ser realizado usando a rotina *clone* disponível para todos objetos Eiffel. Isto pode ser feito da seguinte maneira:

```
seu_carro:=clone(meu_carro)
```

A semântica da rotina *clone* está mostrada na figura 3.5.

#pg049#nt090#cm00#

Semântica da clonagem

3.8 Operadores básicos com exemplos

Operador de igualdade (=): Dois objetos são iguais se eles estão ligados à uma mesma seção de memória. Se alguém quiser testar se os objetos *x* e *y* são iguais, uma expressão do tipo *x=y* poderia ser usada.

Exemplo:

```
meu_valor,seu_valor : REAL
if meu_valor=seu_valor then –- se os valores são os mesmos tome
                                                    alguma atitude
   -- alguma atitude
end
```

Operador de desigualdade(/=): Para testar se os objetos *x* e *y* não são iguais, uma expressão do tipo *if x/=y* poderia ser usada.

Exemplo:

```
meu_valor,seu_valor : INTEGER
if meu_valor/=seu_valor then –- se os valores não são iguais tome
                                                     alguma atitude
   -- alguma atitude
end
```

#pg050#ntnnn#cm00#

#pg051#nt090#cm00#

\\ (Operador binário para resto)

Exemplo:

```
a,b,c : INTEGER
b :=11
c :=4
a :=b\\c --valor é 3
```

< (Operador binário para menor que)

Exemplo:

```
b,c : INTEGER
b :=10
c :=4
if b<c then --realiza alguma ação apenas se b é menor que c
     --alguma ação
end
```

<= (Operador binário para menor ou igual)

> (Operador binário para maior que)

>= (Operador binário para maior ou igual)

Operadores do tipo REAL:

+ (Operador binário para adição)

- (Operador binário para subtração)

* (Operador binário para multiplicação)

/ (Operador binário para divisão)

^ (Operador binário para exponenciação)

< (Operador binário para menor que)

<= (Operador binário para menor ou igual)

> (Operador binário para maior que)

>= (Operador binário para maior ou igual)

Operadores do tipo BOOLEAN:

not (operador unário para negação lógica)

Exemplo:

```
b,c : INTEGER
b:=10
c:=4
if not (b<c) then -- realiza alguma ação apenas se c é
                  -- menor ou igual a b
   --alguma ação
end
```

#pg052#nt065#cm layout errado#

or (operador lógico binário "ou")

Exemplo:

```
b,c: INTEGER
b:=10
c:=4
if b>0 or c>2 then  -- toma a ação abaixo se b é positivo ou c é
maior que 2.
   -- alguma ação
end
```

and (operador lógico "e")

Exemplo:

```
b,c: INTEGER
b:= 10
c:= 4
if b>0 and c>2 then -- toma a ação abaixo se b é positivo e c é
maior que 2.
   -- alguma ação
end
```

implies (usada em assertivas)

or else (operador lógico binário para "ou" em "curto-circuito")

Exemplo:

```
b,c: INTEGER
a:=12
```

```
b:= 10
c:= 40
if b=0 or else c//b > 2 then
-- toma a ação abaixo se b é zero ou c//b é maior que 2
-- avalia a expressão c//b apenas se b não é igual a 0
   -- alguma ação
end
```

and then (operador lógico binário para “e” em “curto-circuito”)

Exemplo:

```
b,c: INTEGER
a:=12
b:=10
c:=40
```

#pg053#nt090#cm00#

```
if b>0 and then c//b>2 then
--execute a ação apenas se b é positivo e c//b é maior que 2
--calcule a expressão c//b apenas se b é positivo
   --alguma ação
end
```

Os últimos dois operadores, “*or else*” e “*and then*”, às vezes são chamados operadores de curto circuito. Para o operador "*or else*", se a primeira expressão é verdade, a segunda expressão nunca é calculada. Para o operador "*and then*" , se a primeira expressão é falsa, a segunda expressão nunca é calculada.

3.9 Ramificação

O tipo mais simples de ramificação é a cláusula *if*. Esta estrutura de controle é usada quando a execução de uma ou mais linhas de um código, um bloco de código, é baseada no resultado de um teste lógico que é executado antes de entrar no bloco de código. O teste lógico requer a avaliação de uma expressão do tipo *boolean*. Tal expressão pode ser verdadeira (*TRUE*)ou falsa (*FALSE*). A forma desta estrutura é:

```
if uma_expressão_booleana then
   comando(s)
end
```

Exemplo:

```
if velocidade_no_solo > 160 then
   --o avião decola
end
```

Outra estrutura simples de controle é a estrutura *if-then-else*. É usada quando uma escolha deve ser feita entre dois blocos de código. A escolha está baseada na avaliação de uma expressão do tipo *boolean.* Esta estrutura de controle é construída como segue:

```
if expressão_booleana then
   bloco_1
else
```

```
   bloco_2
end
```

Aqui, bloco_1 e bloco_2 representam uma ou mais linhas de código.

#pg054#nt040#cm00#

Exemplo:

```
if velocidade_em_terra > 160 then
   -- o avião decola
else
   -- o avião para e a decolagem é interrompida
end
```

O comando **if-then-else** podem ser "aninhado". Considere o segmento de código seguinte:

```
if expressão1 then
   comando1
else
   if expressão2 then
      comando2
   else
      comando3
   end
end
```

Se expressão1 for verdadeira, então comando1 será executado. Caso contrário, se expressão2 for verdadeira será executado o comando2; caso contrário será executado o comando3.

Suponha que um entre vários "ramos" devesse ser executado baseado na avaliação de expressões que podem ser falsas ou verdadeiras. O **comando if-elseif-else** pode ser adequado.

Esta construção é a seguinte:

```
if expressão1 then
   comando1
elseif expressão2 then
   comando2
elseif expressão3 then
   comando3
else
   comando4
end
```

A expressão *else* na construção acima é opcional. Não existe um limite para o número de expressões de *elseif*.

Exemplo:

```
if velocidade_em_terra > 220 then
   -- diminuir a velocidade
```

#pg055#nt000#cm00#

#pg056#nt050#cm layout errado#

3.10 Iteração (loop)

Iteração ou loop é uma operação lógica fundamental da computação. Um único comando ou, mais tipicamente, um bloco de comandos são executados repetidamente, até que alguma condição de parada seja satisfeita. Se a condição de parada nunca for satisfeita, a execução das declarações dentro do loop continuará indefinidamente e o programa tipicamente parece "travar" (jargão que significa que o programa parece não estar fazendo qualquer coisa de útil uma vez que ele não exibe resultados).

Num loop corretamente construído, a condição de parada é eventualmente encontrada.

A forma geral das construções de iteração é:

```
From
   instruções_de_inicialização
Until
   condições_de_saída_do_loop
loop
   corpo_do_loop
end
```

O comando (ou comandos) que compreendem as *instruções_de_inicialização* são executados exatamente uma vez. As *condições_de_saída_do_loop* são testadas antes de cada execução do loop. Se a expressão for avaliada como falsa, o loop é executado; de outra maneira o loop é terminado e o controle é transferido para a linha abaixo da declaração end. Claramente, algumas ações são encontradas no corpo do loop (os comandos entre palavras loop e end) que eventualmente tornarão as *condições_de_saída_do_loop* verdadeiras.

Por exemplo, imagine que queremos mostrar todos os inteiros que são potência de 2 até 65536. O segmento do código que realiza isto usando o comando loop é assim construído:

```
from index:=1
until index=65536
loop
   index:=index * 2
   io.put_int(index)
   io.new_line
end
```

A instrução index:=1 representa as *instruções_de_inicialização*. Ela faz com que index assuma o valor inicial de 1. A declaração index=65536 representa as *condições_de_saída_do_loop* e as três linhas do código logo abaixo de loop representam o *corpo_do_loop*.#pg057#nt090#cm00# O comando *index := index * 2* substitui o valor antigo de *index* por um valor duas vezes maior. O comando deve ser lido assim: "index recebe index vezes 2".

Nós examinaremos outros exemplos para ilustrar o uso do *loop*.

No próximo exemplo suponha que nós queiramos computar a soma de uma série.

```
1+2+3+4+5+6+....+1,000,000
```

Um segmento de código do Eiffel para computar essa soma é dado abaixo. As várias partes da construção do *loop* são mostradas e comentadas.

```
indice, soma : INTEGER
from
   -- comandos de inicialização do loop
   soma:=0
   indice:=0
until indice=1000000 -- condição que encerra o loop
loop
   -- corpo do loop
   indice:=indice+1
   soma:=soma+indice
end
io.put_string("Soma = ")
io.put_int(soma)
io.new_line
```

A instrução indice:=indice+1 é lida como "índice recebe índice mais 1"

As instruções de inicialização, *soma:=0* e *indice:=0*, é correta porém desnecessária. Ambos os objetos assumem o valor 0 em virtude de sua declaração. Incidentalmente, para aqueles que estão interessados, a soma mostrada na tela é 1784293664.

Em uma outra aplicação vamos aproximar a bem conhecida série geométrica: $1 + 1/2 + 1/4 + 1/8 + ... + (1/2)^n$ cujo valor teórico equivale a 2. Nós queremos continuar adicionando números até que o próximo número da série seja equivalente ou menor que 10^{-9}. O segmento de código a seguir usa uma construção de *loop* para aproximar a soma.

```
proximo_termo, soma  : REAL
from
   soma:=1.0
```

#pg058#nt080#cm layout errado#

```
   proximo_termo:=0.5
until  proximo_termo < 0.000000001
loop
   soma:=soma +  proximo_termo
   proximo_termo:=proximo_termo/2.0
end
```

A parte de inicialização do loop altera o valor de soma para 1.0 e *proximo_termo* para 0,5. No corpo do *loop*, *soma* é aumentada pelo valor corrente de *proximo_termo*. A seguir, o valor de *proximo_termo* é alterado para metade de seu valor anterior.

A soma realizada pelo código acima é igual a 2.

Muitos exemplos adicionais de construções com *loop* aparecerão mais tarde em listagens de programa.

3.11 Rotinas

Rotinas existem em duas formas: comandos e consultas. Rotinas são utilizadas quando um objeto recebe um comando ou consulta. Por exemplo, o comando *meu_ponto.alterar_coordenada_x(50)* troca o atributo *x* de *meu_ponto* para um valor igual a 50. A consulta *meu_ponto.angulo* computa a coordenada polar do *angulo* de *meu_ponto*.

Comandos tipicamente mudam o estado interno do objeto que eles estão utilizando. Consultas nunca mudam o estado interno do objeto que eles estão utilizando. Uma consulta corretamente construída retorna informações do objeto, sem modificar seu estado. Embora a linguagem Eiffel permita alguém definir uma rotina que mude o estado interno e também retorne as informações de um objeto, esta prática é totalmente desaconselhável.

Comandos e consultas são especificados em uma seção de características de uma classe (feature). Tanto a informação de interface quanto os detalhes de implementação são dados. O usuário de uma classe (consumidor) precisa acessar somente a parte de interface da rotina. O criador da classe precisa acessar os detalhes de implementação quando realiza manutenção de rotina. (Manutenção é uma atividade que ocorre quando (1) erros são apresentados e corrigidos, (2) melhoramentos em capacidade são desejados, (3) melhorias em eficiência são procuradas).

Nós consideraremos somente a mais simples estrutura para uma rotina neste capítulo. Em capítulos mais avançados outros componentes de uma rotina, como *pré* e *pós-condições*, serão discutidos.#pg059#nt075#cm layout errado# A estrutura de sintaxe de uma simples rotina é a seguinte:

```
nome_rotina [(lista_opcional_de_parâmetros)]
[:tipo_opcional_de_retorno] is
local
   declaração_de_objetos
do
   -- corpo da rotina
end –- nome_rotina
```

O nome da rotina (o mesmo *nome_rotina*) deve ser cuidadosamente escolhido. Ele deve descrever o propósito da rotina. Para uma rotina comando, deve ser usado um verbo. Para uma rotina consulta, deve-se usar um substantivo que descreve aquilo que é retornado. Por exemplo: uma rotina comando para alterar o peso de um carro pode ter o nome *mudar_peso*. Se outra rotina tem como função calcular o volume de um carro, seu nome pode ser *volume*.

A lista de parâmetros, se presente, contem as informações que deve ser colocadas para usar esta rotina.

O tipo de retorno, se presente, indica o tipo de informação que é computada e retornada para quem chamou a rotina (a rotina que chamou a função).

A declaração de objetos, é uma lista de nomes de objetos seguidos de seus respectivos tipos de classe. Lembre-se que a criação ou inicialização de um objeto não resulta de sua declaração (exceto para seus tipos básicos INTEGER, CHARACTER, REAL e BOOLEAN). Cada objeto assume um "valor" padrão de Void até que o objeto seja criado explicitamente por um programador a não ser que seja de um tipo básico.

As instruções contidas entre os delimitadores *do* e *end* representam o corpo ou detalhes de implementação da rotina.

Para ilustrar o conceito de comando e consulta, nós construiremos uma classe simplificada IMPOSTO cujos detalhes são mostrados na listagem 3.4.

O atributo *renda_dedutivel* pode ser consultado mas não modificado. Ele tem semântica "read-only". Se alguém quiser mudar o valor deste atributo, somente o comando *criar* pode ser usado para este propósito.

Listagem 3.4 Classe IMPOSTO para ilustrar os comandos e constultas

```
class IMPOSTO
creation criar
feature
      #pg060#nt080#cm00#
   -- Atributo de consulta
   renda_dedutivel; REAL
   -- Criação e comando ordinário
   criar(quantia: REAL) is
      do
         renda_dedutivel := quantia
      end
   -- Função consulta
   imposto_devido: REAL is
   -- Computação do imposto baseada no atributo renda_dedutivel
      do
         if renda_dedutivel<6000.0 then
            Result:=0.0
         elseif renda_dedutivel<22000.0 then
            Result:=0.15 * renda_dedutivel
         else
            Result:=3300.0 + 0.28 * (renda_dedutivel-22000.0)
         end
      end
end --class IMPOSTO
```

Na função *imposto_devido*, o imposto devido é $0 se a *renda_dedutivel* é menor que $6000, é 15% da *renda_dedutivel* se o rendimento está entre $6000 e $22000 e é $3300 mais 28% do excesso de 22000 da *renda_dedutivel*, quando a renda ultrapassa 22000.

O comando *criar* pode ser usado com uma rotina de criação ou como um comando ordinário. O valor passado altera o valor corrente do atributo *renda_dedutivel*.

A listagem 3.5 exibe uma simples aplicação que exercita a classe IMPOSTO.

Programa teste simples para classe IMPOSTO.

```
class APLICAÇÃO
creation comece
feature

     #pg061#nt080#cm00#
comece is
   local
   meu_imposto:IMPOSTO
   do
      -- use o comando criar para criar e inicializar o objeto
meu_imposto
      !!meu_imposto.criar(40000.0)
      io.put_string("Os impostos a pagar por $")
      -- use o atributo renda_dedutivel para retornar informação
      io.put_real(meus_impostos.renda_dedutivel)
      io.put_string(" = $")
      -- use a rotina imposto_devido para retornar informação
      io.put_real(meu_imposto.imposto_devido)
      io.new_line
      meu_imposto.make(200000.0)
      io.put_string("Os impostos a pagar por $")
      io.put_real(meu_imposto.renda_dedutivel)
      io.put_string(" = $")
      io.put_real(meu_imposto.imposto_devido)
      io.new_line
   end
end -- classe APLICAÇÃO
```

As duas classes IMPOSTO e APLICAÇÃO constituem uma aplicação completa. O objeto meu_imposto é criado e inicializado com o valor 40000 usando o comando de criação *criar* com parâmetro 40000. A renda dedutível atual é acessada diretamente usando o atributo de consulta *renda_dedutivel*. Esse valor pode ser lido mas não alterado diretamente. A função de consulta *imposto_devido* é usada para acessar o imposto devido para a renda dedutível atual.

O valor da renda dedutível é alterado para 200000 usando o comando *criar* com o parâmetro 200000. Então o atributo e a função de consulta são usados para mostrar novos dados sobre o imposto.

3.12 Vetores (*Arrays*)

Vetores são usados para armazenar uma coleção de elementos "similares". O significado de "similar" será mostrado brevemente. Cada elemento do vetor tem um endereço único dito ser seu índice, um valor inteiro. Através do índice alguém pode inserir um elemento em um local único ou acessar um elemento de um local único no vetor.

#pg062#nt080#cm00#

Algumas linguagens fornecem o vetor como um tipo básico enquanto outras linguagens, tal como Eiffel, fornecem uma biblioteca externa como suporte para vetores. Os vetores em Eiffel são fornecidos, através de uma classe padrão chamada ARRAY.

A figura 3.6 mostra um vetor de elementos com um índice que varia de 1 a 5.

Um vetor de elementos.

Qual a natureza dos objetos em um vetor como o representado na figura?

No contexto, programação orientada por objeto (nosso contexto) os elementos são objetos. Cada objeto é uma instância de uma classe "base" ou uma de suas classe descendentes. Nesse sentido os objetos são "semelhantes" em relação à seu tipo. Muitas vezes os elementos nos vetores são do mesmo tipo.

Uma típica declaração de um vetor é:

```
meu_vetor: ARRAY [ALGUM_TIPO]
```

onde ALGUM_TIPO é o tipo "base" ao qual pertencem todas as instâncias de objetos.

O comando *put* da classe ARRAY, para inserir um objeto, *meu_objeto*, num índice especificadoé:

```
meu_vetor.put(meu_objeto,índice)
```

Na expressão acima *meu_vetor* é o nome do vetor onde *meu_objeto* é inserido. Logicamente, *índice* é um determinado inteiro.

A figura 3.7 retrata uma introdução do *meu_ojeto* como o quarto dos cinco elementos do vetore mostrado na figura.

#pg063#nt080#cm00#

```
meu_vetor.put(meu_object,4)
```

O comando *put.*

A função de consulta da classe ARRAY, *item*, que acessa um objeto em uma posição específica é:

```
meu_vetor.item(indice)
```

É claro que *indice* deve ser um valor inteiro dentro dos limites legais para um dado vetor. O vetor na figura 3.7 tem um limite legal de 1 até 5.

A figura 3.8 mostra o elemento na posição 4 sendo acessado com a consulta *item.*

```
meu_vetor.item(4)
```

A consulta *item.*

A rotina de criação da classe ARRAY, *make*, para a construção de uma instância da classe ARRAY é:

```
!!meu_vetor.make (limite_inferior, limite_superior)
```

Os valores inteiros *limite_inferior* e *limite_superior* especificam a variação legal do índice do vetor. Após criar o vetor, *meu_vetor*, os elementos em cada posição assumem seus valores padrão (provavelmente *Void*).

#pg064#nt090#cm00#

Nós ilustraremos todas as idéias anteriores construindo um vetor de veículos. Haverá três tipos de veículos no vetor: CARRO, AVIÃO, e BARCO. Nenhum dos detalhes das três classes será mostrado com exceção de que todos as três são subclasses da classe VEÍCULO.

```
class VEÍCULO
--Classe básica. Nenhum detalhe mostrado.
end -- classe VEÍCULO

class CARRO
inherit
   VEÍCULO
-- Nenhum detalhe mostrado.
end  -- classe CARRO

class AVIÃO
inherit
   VEÍCULO
-- Nenhum detalhe mostrado.
end -- classe AVIÃO

class BARCO
inherit
   VEÍCULO
-- Nenhum detalhe mostrado.
end –- classe BARCO

class APLICAÇÃO

creation
   inicio

feature

   inicio is
      local
      meu_vetor: ARRAY [VEÍCULO]  -- Tipo básico é VEÍCULO
      meu_carro: CARRO
      meu_barco: BARCO
      meu_avião: AVIÃO
      do
         !!meu_vetor.make (1,3)
         !!meu_carro
```

#pg065#nt090#cm00#

```
         !!meu_barco
```

```
          !!meu_avião
          meu_vetor.put (meu_carro, 1)
          meu_vetor.put (meu_barco, 2)
          meu_vetor.put (meu_avião, 3)
       end
end  --classe Aplicação
```

A figura 3.9 descreve a construção de *meu_vetor* contendo três veículos.

Depois de:
```
!!meu_vetor.make (1,3)
```

Depois de:
```
!!meu_vetor.put(meu_avião, 3)
```

Vetor de três veículos

#pg066#nt095#cm00#

Nós depois consideraremos o problema da ordenação de um vetor de números inteiros de tamanho 3. Nós retornaremos o assunto da ordenação no Capítulo 4 quando nós o apresentaremos de uma maneira mais séria e discutiremos diversos métodos importantes de ordenação.

Esta simples aplicação nos permitirá rever o assunto das rotinas de ramificação e seus parâmetros assim como vetores.

Suponha que nós declaremos uma vetor como a seguir.

```
local
   dados : array[INTEGER]
do
   !!dados.make(1,3)
```

Agora nós queremos inserir valores inteiros nas posições 1, 2 e 3. Finalmente nós queremos reordenar os números no vetor colocando o menor na posição 1, o segundo menor na posição 2 e o maior na posição 3. Este processo é chamado ordenação.

A listagem 3.6 nos apresenta uma rotina de ordenação que executa esta missão.

Ordenação uma vetor de 3 números inteiros.

```
class APLICAÇÃO_ORDENACAO

creation
   comece

feature

   comece is
      local
         dados : array[INTEGER]
      do
         !!dados.make(1,3)
         dados.put(30,1)
         dados.put(5,2)
```

```
            dados.put(25,3)
            ordene_3(dados)
            mostre(dados)
        end

    ordene_3(dados : array[INTEGER]) is
    require
        vetor_de_tamanho_certo : dados.count=3
```

#pg067#nt090#cm layout errado#

```
    local
    temporário: INTEGER
    do
        if dados.item(1)>dados.item(2) and dados.item(1) >
dados.item(3) then
            temporário:= dados.item (3)
            dados.put (dados.item (1),3)
            dados.put (temp,1)
        elseif dados.item(2)>dados.item(1) and
dados.item(2)>dados.item(3) then
            temporário:= dados.item (3)
            dados.put (dados.item (2),3)
            dados.put (temp,2)
        end
        if dados.item(1) > dados.item(2) then
            temporário:= dados.item (2)
            dados.put (data.item (1),2)
            dados.put (temp,1)
        end
    end

    imprimir (os_dados:ARRAY [INTEGER]) is
    local
        índice: INTEGER
    do
        from
            índice := 0
        until
            índice = os_dados.count
        loop
            índice := índice + 1
            io.put_int (os_dados.item (índice))
            io.put_string (“ ”)
        end
        io.new_line
        end

end -- classe APLICACAO_ORDENACAO
```

A primeira ordem de execução da rotina *comece* é a criação do vetor *dados* com índice inferior igual a 1 e superior igual a 3 (índice entre 1 e 3). A seguir, os valores 30, 5, e 25 são inseridos no vetor. A figura 3.10 descreve o vetor *dados* depois desse passo.

#pg068#nt095#cm00#

Vetor inicial de três inteiros que serão ordenados.

A rotina *ordene_3* é a próxima a ser solicitada com *dados* enviado como parâmetro. A cláusula *require* representa uma pré-condição que deve ser satisfeita no momento em que a rotina for solicitada. Pré condições serão discutidas com mais detalhes no capítulo 6.

O primeiro teste que é realizado determina se, o inteiro do índice 1 é maior do que o inteiro do índice 2 e do índice 3 (em outras palavras, o inteiro do índice 1 é o maior entre os três inteiros). Se isso ocorrer, como ocorre neste caso, o inteiro do índice 1 e o do índice 3 são trocados, colocando então o maior dos três inteiros no índice 3. A figura 3.11 mostra como ficou o vetor depois dessa operação de troca.

Vetor de três inteiros depois da primeira troca.

Finalmente os inteiros do índice 1 e do índice 2 são comparados. Se o inteiro do índice 1 for maior que o inteiro do índice 2 eles serão trocados. É o que ocorre neste caso. A figura 3.12 mostra como ficou o vetor depois dessa última operação de troca.

Vetor de três inteiros depois da última operação de troca.

#pg069#nt085#cm layout errado#

Agora os números estão sortidos. Você pode provar que maneira deste "algoritmo"(uma série de operações que realizam uma série de serviços) sempre armazenarão os três números conforme as condições?

A rotina *display* mostra os valores dos números crescentes nas posições 1, 2 e 3. A consulta *count* é usada para adquirir o maior número do array os_dados.

3.13 Cadeias de caracteres (Strings)

Em programação, uma *string* é um vetor de caracteres, não é algo com que você amarra seus sapatos. Estes caracteres podem ser letras maiúsculas ou minúsculas, números ou símbolos do teclado como '$', '%' ou '&'. Até um espaço em branco é um caracter.

Por causa dos vetores baseados em caracter, as *strings*, serem tão importante em programação, nós os estudamos como um tipo especial de vetor. Em Eiffel a classe STRING armazena as propriedades deste importante tipo de dados.

Em programas nós usamos *strings* para nomes de pessoas ou coisas. Nós, tipicamente, tratamos uma *string* como uma simples entidade única apesar dela ser constituída de vários caracteres. Então, a *string* com a sequência de caracteres 'M', 'a', 'r', 'i' e 'a' geralmente será vista como uma única entidade "Maria" mais propriamente do que os caracteres individuais. É claro que os caracteres individuais de uma *string* podem ser acessados usando a consulta *item* de *array* discutido na sessão anterior.

Uma *string* é uma seqüência de caracteres delimitados por um par de aspas. Alguns exemplos de *strings* são fornecidos abaixo.

Exemplos de Strings

```
“Meu nome é Richard Wiener.”
“Testando, testando, 1, 2, 3”
“!@#$%^&*()-+”
“                          ”
```

O leitor pode desejar parar e considerar as operações que poderiam ser desejáveis em uma *string*. Certamente as operações mais básicas poderiam incluir (1) criação de uma *string* de um determinado tamanho – um vetor de caracteres que pode suportar um número pré-determinado de caracteres, (2) inserir caracteres na *string* em diferentes locais, (3) preencher toda a *string* usando uma atribuição de outra *string* (uma seqüência de caracteres limitada por aspas, como “oi” ou “tchau”), (4) acessar caracteres de uma posição específica, (5) copiar uma *string* para outra (a que vai ser copiada já deve ter sido iniciada) e (6) retornar o tamanho de uma *string* – o número de caracteres atuais da *string*, não o seu total.

#pg070#nt095#cm00#

Existem operações adicionais que você pode desejar acrescentar a essa lista relativamente pequena? Essa é a questão que o desenvolvedor da classe STRING deve se perguntar quando está construindo esse componente de software reutilizável.

Surpreender-te-ia saber que a classe STRING contida na biblioteca do Eiffel contém aproximadamente 60 rotinas que definem o comportamento (comandos e consultas) de um objeto STRING? A fim de apreciar o trabalho realizado no desenvolvimento de um importante componente de software reutilizável como esse, vamos examinar a funcionalidade da classe STRING um pouco antes de olhar a sua interface formal e demonstrar seu uso em uma aplicação.

A classe STRING é dividida em várias seções *feature*, cada uma contendo um conjunto logicamente relacionado de operações. Na tabela 3.1, as várias seções *feature* são mostradas com seus propósitos e uma lista de algumas das rotinas nas seções *feature* são listadas com uma breve descrição de seus propósitos.

Tabela 3.1 Análise da classe STRING

(1) **Acesso** – Usado para obter várias partes do objeto STRING
- **has** – a *string* inclui um caracter em particular?
- **index_of** – posição da primeira ocorrência de um caracter
- **item** – caracter em índice específico
- **item_code** – código numérico de caracter em índice específico
- **substring_index** – posição de ocorrência de outra *string* contida na *string* dada.
- operador “**@**” – caracter em índice específico (alternativo à item)

(2) Comparação – Usado para comparar duas *strings*
- **is_equal** – as duas *strings* contém a mesma seqüência de caracteres?
- operador “<” – uma *string* é lexicograficamente menor que outra?

(3) Conversão – Usado para converter *string* de uma forma para outra
- **mirror** – inverte a ordem dos caracteres na *string*
- **mirrored** – teste para ver se outra *string* é espelho da primeira
- **to_double** – converte para o tipo DOUBLE, se possível
- **to_integer** – converte para o tipo INTEGER, se possível
- **to_lower** - converte todos os caracteres maiúsculos em minúsculos

to_real – converte para o tipo REAL, se possível
to_upper – converte todos os caracteres minúsculos em maiúsculos

#pg071#nt095#cm00#

(4) Duplicação **–** usada para copiar partes de uma *string* para outra
substring – cópia de uma *substring* (*string* contida em uma outra *string*) contendo caracteres entre um índice e outro

(5) Mudança de elementos **–** usada para modificar partes de uma *string*
append – adiciona uma cópia de uma *string* para o final de outra
copy – transfere caracteres da *string* fonte para a *string* destino
extend – adiciona um caracter no fim de uma determinada *string*
fill_blank – preenche uma *string* com caracteres em branco
head – remove todos, menos os *n* primeiros caracteres
insert – adiciona uma *string* à esquerda do índice especificado em uma determinada *string*
left_adjust – remove todos os espaços iniciais em uma *string*
precede – adiciona um caracter na frente de uma determinada *string*
prepend – adiciona uma *string* na frente de uma determinada *string*
put – substitui um caracter em um índice especificado por um determinado caracter
replace_substring – copia caracteres de uma outra *string* para posições específicas da *string* dada
replace_substring_all – substitui todas as ocorrências de certa *string* por novas
right_adjust – remove todos os espaços finais de uma determinada *string*
set – impróprio para discutir aqui
share – faz com que a *string* atual compartilhe o texto de outra *string*; qualquer mudança no texto da outra *string* afetará o original
tail – remove todos os caracteres, exceto os *n* últimos de uma determinada *string*

(6) Inicialização **–** usada para criar um objeto da classe STRING
make – aloca espaço para no mínimo *n* caracteres

(7) **Medição –** usada para obter aspectos numéricos de uma *string*
capacity – espaço alocado
count – número atual de caracteres em uma *string*
occurrences – número de vezes que um caracter específico aparece em uma *string*

(8) Saída de dados **–** usada para escrever uma *string*
out – cria uma representação que pode ser impressa

(9) Remoção **–** usada para remover partes de uma *string*
prune – remove a primeira ocorrência de um caracter específico
prune_all – remove todas as ocorrências de um caracter específico
remove – remove o *i*-ésimo caracter
wipe_out – remove todos os caracteres

(10) Redimensionamento **–** usada para mudar dinamicamente o tamanho de uma *string*
adapt_size – muda o tamanho para acomodar o número atual de caracteres

grow – garante que a capacidade é pelo menos o número especificado
resize – realoca espaço para acomodar um número específico de caracteres

(11) Relato de Status **–** usada para obter algumas características importantes de uma *string*
consistent – a *string* dada pode ser o "destino" de uma operação de cópia?

#pg072#nt000#cm00#

#pg073#nt070#cm00#

```
to_real: REAL
   --valor real;
   --por exemplo, quando aplicado à "123.0", produzirá 123.0
to_upper
   --converte para letras maiúsculas
substring (n1, n2: INTEGER): like Current
   --copia uma substring contento todos os caracteres do índice n1
ao n2
append(s:STRING)
   --junta uma cópia de 's' no fim.
copy (outro: like Current)
   --reinicializa copiando os caracteres de outro
   --(isto é usado também pelo 'clone')
fill_blank
   --preenche com espaços
insert(s: like Current; i:INTEGER)
   --adiciona 's' à esquerda da posição 'i' na string atual
put(c: CHARACTER; i: INTEGER)
   --Substitui um caractere da posição 'i' por 'c'
capacity: INTEGER
   --quantidade de memória alocada
count :INTEGER
   -- número de caracteres de uma string
occurences (c: CHARACTER):INTEGER
   --numera quantas vezes que 'c' aparece na string
wipe_out
   --remove todos caracteres
end --classe STRING
```

O apêndice 1 fornece todas as informações que são requeridas para o uso da classe STRING numa dada aplicação. Como parte do processo de aprendizagem para se tornar um consumidor competente, e o leitor deve estudar cuidadosamente as informações da interface deste apêndice.

Uma estratégia altamente recomendável para se tornar familiar e confortável com o uso de uma classe tal como STRING é criar um programa teste que exercite algumas destas funções. Apesar disto poder tomar algum tempo e mesmo parecer tedioso, a recompensa será grande.#pg074#nt000#cm00#

#pg075#nt080#cm00#

```
    else
       io.put_string ("str1/=mensagem")
    end
```

```
        io.new_line

        --Muda o primeiro caracter da string str1 e então compara
        --str1 e mensagem novamente
        str1.mirror
        io.put_string (“str1=  “)
        io.put_string (str1)
        io.new_line

        --Inverte a seqüência de caracteres na string str1
        str1.mirror
        str1.to_upper
        io.put_string  (“str1= “)
        io.put_string  (str1)
        io.new_line

        --Obtém a primeira ocorrência da letra ‘G’
        io.put_string (“Primeira ocorência de ‘G%’=”)
        io.put_int (str1.index_of (‘G’,1))
        io.new_line

        --Obtém a primeira ocorrência da substring “ING”
        io.put_string (“Primeira ocorência de ‘ING%’=”)
        io.put_int (str1.substring_index of (‘ING’,1))
        io.new_line

        --Escreve o valor da string true_constant
        io.put_string (str1.true_constant)
        io.new_line

        --Atribui a substring de str1 entre os índices
        --24 até 27 para a string str2
        str2:= str1.substring (24,27)
        io.put_string (“str2= ” )
        io.put_string (str2)
        io.new_line

        --Atribui str2 ao valor real r depois de converter a
string para real
        r:= str2.to_real
        io.put_string (“r= ”)
```

#pg076#nt000#cm00#

#pg077#nt000#cm00#

#pg078#nt065#cm00#

Como a entrada de dados é obtida no programa?

A dispositivo de entrada padrão é o teclado. O segmento seguinte de código ilustra como se poderia entrar com um valor real, um valor inteiro, e um valor de caracter.

```
meu_inteiro: INTEGER
meu_caracter: CHARACTER
meu_real: REAL
io.readint  -- comando de leitura
io.readchar -- comando de leitura
io.readreal -- comando de leitura

meu_inteiro:=io.lastint -- consulta
meu_caracter:=io.lastchar -- consulta
meu_real :=io.lastreal –- consulta
```

Os comandos de entrada usados são *readint, readchar, e readreal*. Cada um é transmitido através do objeto padrão de entrada e saída (*io*). Os valores que de entrada (informação datilografada no teclado) são obtidos pelas consultas *lastint, lastchar* e *lastreal*, novamente, transmitidos pelo objeto padrão de entrada e saída (*io*). O protocolo rigoroso de separação entre comados e consultas é observado aqui. Embora possa ser tentador fazer *readint* retornar um inteiro, o ponto de vista examinado aqui é que o comando *readint* muda o estado da situação interna do objeto *io* e a consulta *lastint*, retorna parte de seu estado interno.

Existem muitos comandos de entrada e saída de comandos além de consultas definidas na classe STD_FILES de Eiffel que padronizam as entradas e saídas. Uma porção da interface desta classe é dada na listagem 3.9.

Porção da interfase para classe STD_FILES

```
class interface STD_FILES
    feature -- muda elemento
       new_line
          -- Escreve uma mudança de linha na saída padrão

       put_char(c: CHARACTER)
          -- Escreve ´c´ no final da saída padrão.

      #pg079#nt095#cm00#
       put_double (d: DOUBLE)
       --escreve ‘d’ no final da saída default

       put_int (i: INTEGER)
       -- escreve  ‘i’ no final de saída default

       put_real (r:REAL)
       --escreve ‘r’ no final de saída default

       put_string (s:STRING)
       --escreve ‘s’ no final de saída default

    feature -- entrada de dados

       next_line
       --move para a próxima linha na entrada padrão
```

```
        readchar
        --lê um novo caracter da entrada padrão
        --Disponibiliza o resultado em ‘lastchar’

        readdouble
        --lê um novo double da entrada padrão
        --Disponibiliza o resultado em ‘lastdouble’

        readint
        --lê um novo inteiro da entrada padrão
        --disponibiliza o resultado em ‘lastint’

        readline
        --lê uma linha da entrada padrão
        --disponibiliza o resultado em ‘laststring’

        readreal
        --lê um novo real da entrada padrão
        --disponibiliza o resultado em ‘lastreal’

        readstream (nb_char: INTEGER)
        --lê uma string de no máximo ‘nb_char’ caracteres da entrada
padrão
        --disponibiliza o resultado em ‘laststring’

        readword
        --lê uma nova palavra da entrada padrão
        --disponibiliza o resultado em ‘laststring’

    feature – relatório  de status
```