Uma Experiência de Modelagem de Dados Orientada a Objetos

Roberto Cássio de Araújo

Dissertação Apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
para a Obtenção do Grau de
Mestre em Matemática Aplicada

Área de Concentração: **Ciência da Computação**

Orientador: **Prof. Dr. Routo Terada**

Este trabalho recebeu apoio financeiro do CNPq

– São Paulo, março de 1995 –

Uma Experiência de Modelagem de Dados Orientada a Objetos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Roberto Cássio de Araújo
e aprovada pela comissão julgadora.

São Paulo, 25 de maio de 1995.

Banca examinadora:

- Prof. Dr. Routo Terada (orientador) – IME-USP
- Prof. Dr. Valdemar Waingort Setzer – IME-USP
- Prof. Dr. Hans Kurt E. Liesenberg – DCC-UNICAMP

Agradecimentos

Um trabalho desta natureza é resultado da combinação de diversas entidades que blá blá blá blá... ... o pessoal da biblioteca blá blá... ... o departamento de computação blá blá.... ... blá, sem esquecer os...blá blá. Por outro lado, blá blá.... ... e também ao pessoal do CEC que, além de tudo, ofereciam um dos poucos espaços suficientemente amplos para minha expansão intelectual.

Finalizando, sinto-me obrigado a mencionar nominalmente aqueles que, por merecerem uma menção nominal, serão mencionados nominalmente:

Ao Routo, por tudo e pela paciência.

Àqueles poucos (como Renata, Flávio, Fábio, Léo), que esqueciam as numerocracias e transformavam a matemática em uma ciência humana.

Ao Sol e à Lua, que ignoravam minha forçada indiferença e apareciam pontuais e cotidianos como sempre.

Ao Velvet Underground e Cowboy Junkies, que gravaram a trilha sonora oficial deste trabalho.

E a três musas que eu sempre sentia sempre perto:

Antonieta, Loreley e Pituca.

Aos que não atrapalharam, também!

Muito obrigado.

ao fã clube do

O. Yeah da Silva

Resumo

O objetivo deste trabalho é apresentar o modelo de desenvolvimento de programas chamado de Desenvolvimento de Orientado a Objetos (DOO). Para isto, iniciamos caracterizando o problema de desenvolvimento de programas dando ênfase às duas abordagens mais comuns: desenvolvimento baseado nas instruções que o computador deve executar e desenvolvimento baseado nos dados que o computador deve manipular. A seguir, introduzimos o modelo de programação orientada a objetos destacando cinco de suas propriedades fundamentais (abstração, modularidade, encapsulamento, herança e polimorfismo) e os recursos oferecidos que possibilitam a reutilização de software.

Somente depois desta introdução estudaremos o DOO especificamente, abrangendo as seguintes atividades envolvidas na produção do software: identificação de objetos e seus relacionamentos, definição da iterface da classe e finalmente, sua implementação. Este trabalho é concluído com o desenvolvimento de um programa usando as técnicas apresentadas nos capítulos anteriores.

Abstract

The aim of this work is to introduce the program development model called Object-Oriented Design (DOO). We begin by characterizing the problem of program development, specially the two most common approaches: development based on computer instructions and on data. Then we introduce the object-oriented programming method, pointing out five of its fundamental properties (abstraction, modularity, encapsulation, inheritance and polymorphism) and the features of this method that permit software reuse.

After this introduction we explore DOO, covering the following activities which are part of DOO software production: object identification, class relationship identification, class interface definition and class implementation. We conclude this work with an example of program design using the techniques presented in the earlier chapters.

Conteúdo

Capítulo 1

Introdução

> Somos propensos a falar e pensar de objetos. Os objetos físicos são o exemplo óbvio quando nos dispomos a exemplificar, mas há também todos os objetos abstratos, ou assim se pretende que haja: os estados e qualidades, números, atributos, classes. W.V.Quine, *Falando de Objetos.*

O objetivo deste primeiro capítulo é literalmente "preparar o terreno" para o restante do texto. Para isto, começamos apresentando nosso "universo de discurso", tentando apenas "sugerir" porque o modelo de programação orientada a objetos seduziu uma grande parcela dos programadores e projetistas de sistemas. Para completar, serão apresentadas as primeiras definições e demais convenções usadas no decorrer deste trabalho.

1.1 Problemas, Computadores e Programas

Caracterização 1 *Um computador é uma máquina que processa informação. Este processamento de informação corresponde a manipulação de dados através de um conjunto fixo e limitado de instruções capazes de serem executadas pela máquina.*

Uma vez disponível um computador, podemos usá-lo para resolver problemas. Daí surge um segundo problema: é necessário descrever o processamento que o computador deve executar para resolver o primeiro problema, ou seja, é necessário programar o computador.

Caracterização 2 *Um programa é uma seqüência finita de instruções capazes de serem executadas por um computador.*

Dado um programa que resolve um problema, é importante dispor de algum critério para avaliar sua qualidade. A primeira característica considerada para fazer tal avaliação é seu comportamento de entrada/saída, se ele funciona ou não. No entanto, não acreditamos que este seja o único fator responsável pela qualificação de um programa. Uma sugestão para tal critério é que um programa deve ser:

correto: deve fornecer um resultado correto (segundo suas especificações). Note que as especificações de um programa são fundamentais para avaliar sua correção. Por exemplo, um programa em C++ cujo único comando seja

```
cout << 4;
```

serve para calcular o resultado de uma soma desde que o resultado seja 4.

portátil: deve ser o mais independente possível de uma máquina específica. Desta forma com um número mínimo de adaptações (idealmente nenhuma) será possível transportar um programa que funciona em um tipo de máquina para outro.

eficiente: deve resolver o problema proposto em um tempo mínimo e usando também o mínimo de memória (lembrando que, geralmente, aumentar o desempenho para um destes parâmetros implica na diminuição do desempenho para o outro).

amigável com o usuário: deve oferecer recursos simples e compreensíveis para sua correta operação além de oferecer ajuda quando necessária. Poderíamos incluir também o item *documentação e treinamento* que está, de certa maneira, relacionado com a interface com o usuário: quanto mais simples a interface, menores serão as necessidades destes dois recursos.

baixo custo: deve ter seus custos de projeto, desenvolvimento, operação e manutenção minimizados.

Infelizmente, as instruções executáveis pelo computador são descritas por intermédio de uma notação (a linguagem de programação) nada natural para seres humanos. Como os computadores "exigem" programas para que funcionem, é necessário também treinar uma pessoa para escrever estes programas. O treinamento do programador de computadores torna-se fundamental quando constatamos que a qualidade de um programa é uma conseqüência direta da qualidade do programador que o escreveu.

Caracterização 3 *Um programador é um ser (idealmente humano) que se encarrega de escrever programas de computadores.*

Como conseqüência do avanço tecnológico, os computadores passaram a dispor de recursos (hardware) mais complexos, tornando-se possível resolver problemas cada vez mais complexos com o seu auxílio. Devido ao crescimento da complexidade dos problemas que passaram a ser resolvidos por computador, os programas passaram a ser escritos por equipes de programadores. Surge, assim, a necessidade de uma pessoa para coordenar o trabalho desta equipe.

Caracterização 4 *Um projetista de sistemas é um ser (idealmente humano) que se encarrega de analisar problemas e coordenar o desenvolvimento dos programas que deverão resolver estes problemas. É ele que lida com a complexidade do problema a ser resolvido, tendo como responsabilidade transformar o problema em propostas de programas mais simples que, conjuntamente, resolvem o problema proposto. Quando o desenvolvimento do programa é efetuado por uma equipe de programadores, ele assume também a responsabilidade de coordenar esta equipe.*

Trabalhando em equipe é importante esquecer fatores individuais (como inspiração ou crenças metafísicas) para produzir um resultado (o programa) capaz de ser compreendido ou até mesmo alterado por outras pessoas.

Caracterização 5 *Um método de programação consiste em uma forma de sistematizar o processo de programação através da imposição de roteiros que o programador deve seguir para produzir, no final do processo, um programa capaz de ser executado por um computador.*

Uma vez convencidos que nós, programadores e seres humanos, devemos ser sistemáticos e metódicos, uma questão surge de imediato: *qual é o melhor método de programação?* Uma possível resposta é: *O melhor método de programação é aquele cuja aplicação resulta em melhores programas.*

Mas onde entra...

1.2 Programação Orientada a Objetos

...nesta história toda?

Na verdade, esta é a razão deste trabalho. O bom programa é o resultado de um projeto de software bem sucedido e POO proporciona um bela forma para o programador e o projetista de sistemas alcançarem o almejado sucesso profissional através da utilização de um modelo de programação que produz bons programas. O que esta tese propõe

desde já é que programação orientada a objetos, além de um modelo de linguagens de programação, é um bom método de programação e que um bom método de programação é um bom caminho para se escrever um bom programa.

Booch [Booch91], por sua vez, caracteriza como um projeto de software bem sucedido aquele cujos recursos satisfazem, e possivelmente superam, as expectativas do cliente, que foi desenvolvido dentro do prazo estipulado de forma econômica e que é flexível o suficiente para sofrer mudanças e adaptações. Segundo esta caracterização, ele também propõe que visão arquitetural e ciclo de vida iterativo e incremental são duas características fundamentais nestes sistemas bem sucedidos.

Visão Arquitetural equivale a integridade conceitual.

Boas arquiteturas de software tendem a apresentar diversas características comuns:

- Elas são construídas em níveis de abstração bem definidos.
- Existe uma separação nítida de escopo entre a interface e a implementação de cada nível de abstração, tornando possível a mudança da implementação de um nível sem violar as suposições feitas por seus clientes.
- A arquitetura é simples: comportamento comum é alcançado através de abstrações e mecanismos comuns.

Ciclo de Vida Iterativo e Incremental Projetos orientados a objetos são iterativos no sentido em que eles envolvem o refinamento sucessivo de uma arquitetura orientada a objetos sobre a qual aplicamos a experiência e resultados de cada fase para a próxima iteração de análise e desenvolvimento. São incrementais no sentido em que cada passo através do ciclo de análise/desenvolvimento/evolução leva-nos a refinar gradualmente nossas decisões estratégicas e táticas, convergindo, finalmente, a uma solução que atende às necessidades do usuário final e que, adicionalmente, é simples, confiável e adaptável.

Sumarizando, o objetivo deste trabalho é apresentar e explorar os recursos oferecidos pelo modelo de programação orientada a objetos para facilitar o projeto e desenvolvimento de sistemas computacionais. Nesta exposição buscaremos enfatizar que a adoção deste modelo de programação não se resume na comodidade da simples troca da linguagem com que se programa, mas envolve também uma reformulação completa na forma com que são analisados os problemas e são produzidos os programas de computador.

Para isto, serão apresentados mais cinco capítulos abrangendo os seguintes aspectos de desenvolvimento de programas e programação orientada a objetos:

capítulo 2: apresentar o problema de desenvolvimento de programas, enfatizando dois enfoques possíveis: nas ações que o computador deve executar ou nos dados que

deve manipular. Para completar, é apresentado o conceito de ciclo de vida de um software para avaliar o custo envolvido em seu desenvolvimento.

capítulo 3: apresentar o modelo de programação orientada a objetos através de suas características fundamentais (abstração, encapsulamento, polimorfismo, herança e modularidade) . Embora pareça, à primeira vista, desnecessária tal apresentação (uma vez que "todo mundo" já decorou o significado destes termos), ela é bastante útil porque os novos métodos de desenvolvimento baseados em objetos valem-se exatamente destas propriedades para a criação de novas abordagens de condução do processo de desenvolvimento. Na conclusão deste capítulo apresentamos de que forma estas características de objetos contribuem na concepção de programas que possam ser reaproveitados em outras aplicações.

capítulo 4: finalmente, será estudado o desenvolvimento de programas orientado a objetos. Isto será feito com a exposição de abordagens para resolver os seguintes aspectos relacionados com o desenvolvimento de programas:

- identificar e descrever os objetos componentes do sistema;
- identificar e descrever relacionamentos entre os objetos do sistema;
- implementar os objetos componentes do sistema;
- desenhar a arquitetura e o comportamento do sistema;

Para completar, analisaremos o ciclo de vida de sistemas desenvolvidos com base em objetos.

capítulo 5: para fechar este trabalho, será apresentado um problema que resolveremos com a plicação das idéias expostas no capítulo anterior. Seu objetivo é testar a eficácia do DOO para a solução de um problema real.

1.3 Referências Bibliográficas

Este trabalho teve origem com a leitura de [Mullin89], que descrevia através de um exemplo concreto, como resolver um problema real usando o modelo de objetos. Entretanto, a consulta de [Mona92] mostrou que o problema de desenvolvimento orientado a objetos estava em pleno desenvolvimento e estavam surgindo muitos trabalhos se propondo a resolver a questão, cada um de seu jeito. Finalmente, em [Booch91] foi estudado um método de (análise e) desenvolvimento completo, que propõe tanto as recomendações para dirigir o desenvolvimento como uma notação para retratar os resultados deste desenvolvimento.

Embora estes três trabalhos tenham delineado o formato global deste texto, seu enfoque foi profundamente influenciado por diversas experiências didáticas que justificaram, dentre tantos detalhes, o formato dos capítulos 2 e 3.

Em tempo, o texto que abre este capítulo [Quine89], embora pareça ter sido escrito por encomenda para os defensores de programação orientada a objetos, refere-se especificamente a aspectos de lingüística humana.

1.4 Convenções e Terminologia

Neste texto foi optado por traduzir "object-oriented programming" por *programação orientada a objetos* embora existam outras denominações comuns na língua portuguesa (como "programação orientada para objetos" ou "programação dirigida a objetos"). A utilização de um ou de outro termo fica a cargo do freguês e discussões defendendo a utilização de um ou de outro tendem a ser longas e inúteis.

Como de costume na literatura, usamos a sigla POO para denotar programação orientada a objetos, além de AOO para análise orientada a objetos e DOO para desenvolvimento orientado a objetos. Quando for necessário referirmo-nos a análise e desenvolvimento orientados a objeto conjuntamente, usaremos ADOO. Para completar usaremos simplesmente OO para qualificar alguma linguagem método ou produto como orientado a objetos.

A nomenclatura dos termos relacionados a POO foi tomada da linguagem C++ (embora sejam também utilizados termos como *mensagem*, mais comum em Smalltalk).

Embora tenha buscado tornar este texto independente de linguagens de programação, os exemplos apresentados serão codificados em C++. Por causa desta escolha, os detalhes da implementação de programas exigirão a apresentção de recursos particulares desta linguagem. Quando for necessário mencionar um termo de um programa no meio do texto, usaremos o tipo de caracteres sans serif.

Chamamos de *autor* de um código de programa a pessoa que criou (e implementou) o código, enquanto que um *usuário* desse código é qualquer pessoa que o utiliza em seus programas, ou seja, para um código armazenado no arquivo cod.h, o usuário é aquele que, nos seus programas, escreve a linha #include<cod.h>.

Dizemos que uma propriedade de uma classe é *visível* por um usuário se este tem acesso direto sobre essa característica. No caso de uma propriedade não visível, o usuário poderá manipulá-la apenas por intermédio de funções específicas que realizam esta manipulação ou, em alguns casos, nem precisa ficar ciente da existência desta propriedade.

Capítulo 2

Desenvolvimento de Programas

Neste capítulo será tratado o problema de desenvolvimento de sistemas. Para isto, após uma breve introdução histórica, apresentaremos uma visão geral das duas abordagens mais comuns na atividade de desenvolvimento: a primeira, mais antiga, concentra-se na descriçãoo das operações que o computador deve executar (por isso tratamos este modelo como "programação baseada em ações"); a outra, que vem recentemente conquistando mais adeptos, concentra-se na caracterização dos dados que o computador deve manipular (chamamos este modelo de "programação baseada em dados"). Para terminar, introduziremos o conceito de ciclo de vida de um sistema.

2.1 Introdução

No processo de produção de um software, podemos identificar duas atividades principais: as fases de análise e desenvolvimento do sistema (veja figura 2.1). Embora não exista uma fronteira bem definida separando estas duas atividades, na análise são consideradas as especificações do serviço que o sistema deve oferecer, devendo fornecer como resultado uma descrição do comportamento que o sistema deve apresentar para implementar estas especificações; a partir do problema no mundo real, produz uma representação de seu domínio, um *modelo conceitual*. Na fase de desenvolvimento, é considerado o comportamento desejado do sistema, obtido durante a análise, devendo fornecer como resultado uma descrição detalhada da forma com que é implementado este comportamento, uma representação da solução do problema.

Chamamos de *método de desenvolvimento de sistemas* (ou *método de programação*) a forma com que são conduzidas as atividades de análise e desenvolvimento do sistema. Estes métodos envolvem uma atividade de modelagem do sistema e manipulação do modelo obtido. A função do modelo é abstrair as propriedades relevantes do problema que devem

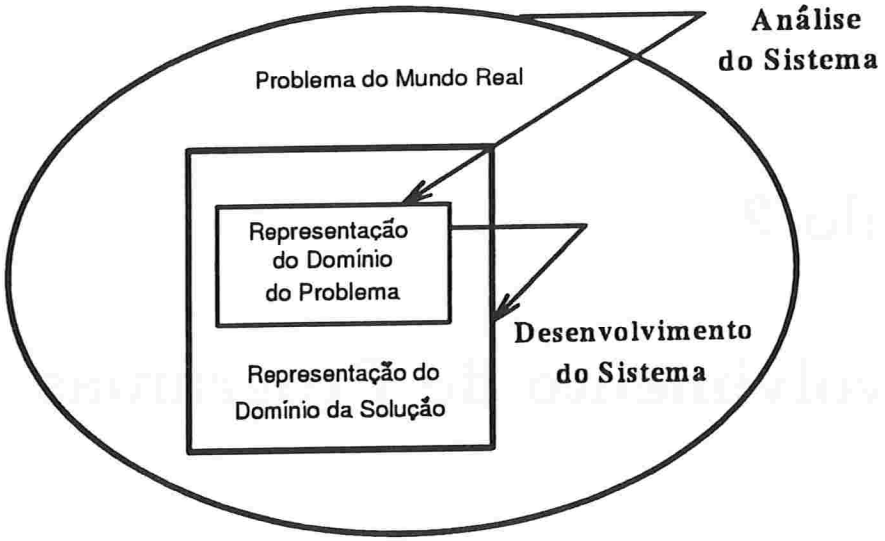

Figura 2.1: Relação entre análise e desenvolvimento

ser consideradas no projeto de programação. Em geral, para facilitar a modelagem, os métodos de análise e desenvolvimento oferecem:

notação: uma linguagem, geralmente gráfica, para exprimir o modelo obtido (notações para descrição de dados, processamento exercido pelo sistema, iteração entre componentes do sistema, etc.)

processo: atividades que conduzem a uma construção ordenada dos modelos do sistema.

ferramentas: artefatos que eliminam o tédio da construção de modelos controlando esta construção de tal forma que erros e inconsistências sejam imediatamente acusados.

Segundo Booch [Booch91], a necessidade de métodos confiáveis para o desenvolvimento de programas decorre do alto grau de complexidade inerente aos próprios programas. Para ele, esta complexidade é decorrente de:

Complexidade do domínio do problema. Os problemas a serem resolvidos freqüentemente envolvem elementos de complexidade inexplicável, nos quais encontramos uma variedade de necessidades concorrentes e até mesmo contraditórias.

Dificuldade de gerenciamento do processo de desenvolvimento. A tarefa fundamental de uma equipe de desenvolvimento é projetar a ilusão de simplicidade dos sistemas de software. Além de facilitar a comunicação dentro desta equipe, o principal desafio de administração é sempre manter a unidade e integridade do desenvolvimento.

Flexibilidade possível através do software. Para permitir sua utilização em diversas áreas de aplicação, as linguagens de programação acabam por oferecer recursos genéricos demais que dificultam sua aplicação para solução de problemas específicos. Por exemplo, considerando o problema de manipulação de base de dados, compare a linguagem C com qualquer outra linguagem de consulta de bancos de dados ou, até mesmo, com Cobol.

Problemas de caracterização do comportamento de sistemas discretos. Em sistemas contínuos, pequenas alterações em suas entradas causam pequenas alterações nas saídas. Sistemas discretos (caso específico de programas de computadores), por sua natureza, assumem um número finito de estados possíveis; em sistemas de grande porte ocorre uma explosão combinatória que torna este número grande demais. Torna-se, conseqüentemente, praticamente impossível prever todos os estados que o sistema pode assumir, o que facilita a ocorrência de erros na modelagem conceitual.

2.2 Métodos de Desenvolvimento de Programas

No período imediatamente após a criação do computador, sua programação tratava-se de uma atividade "artesanal". Foi somente no final da década de 60 que E. W. Dijkstra alertou a comunidade científica para a necessidade de método na atividade de programação de computadores, usando como argumento a incapacidade humana em trabalhar com estruturas muito complexas.

Através do método que foi chamado de *Programação Estruturada*, o próprio Dijkstra apresentava uma forma de sistematizar a decomposição de um programa em programas menores através da decomposição modular do problema. Assim, um programa complexo pode ser dividido em programas mais simples de tal forma que ao se resolver cada um destes problemas simples tem-se "imediatamente" a solução do problema original.

No entanto, este método (e outros derivados dele) sofria do mesmo mal que atacava as linguagens de programação disponíveis na época: limitava-se a organizar a descrição das ações que deveriam ser executadas pelo computador[1]. Estes métodos consistiam basicamente em descrever o funcionamento global do programa e refinar, passo a passo, esta descrição até que o programa completo surgisse naturalmente. Este tipo de enfoque pode ser chamado de "Programação Baseada em Ações": o desenvolvimento é direcionado pela decomposição do problema nas tarefas e subtarefas envolvidas na sua solução.

[1] Até mesmo o termo programação exprime esta natureza: ele denota a descrição seqüencial de uma série de atividades a serem realizadas.

Como uma de suas qualidades, programação estruturada introduzia uma forma eficaz para implementar aquilo que é chamado de *programação modular* (que denota a atividade de divisão de um programa em módulos para permitir a administração do sistema através do gerenciamento de módulos): através da decomposição do problema torna-se imediata a definição dos módulos que resolvem cada parte do problema.

Paralelamente à aceitação de programação estruturada como um método eficaz para o desenvolvimento de sistemas, buscava-se uma forma alternativa de implementar a decomposição modular de um problema: definir cada módulo com base nos dados que o programa manipula. Deste princípio surge o modelo de *Programação com Dados Ocultos*, no qual cada módulo apresenta somente a descrição de uma parte dos dados usados no programa junto com as sub-rotinas que o manipulam de tal forma que toda manipulação efetuada sobre um dado seja implementada por uma função específica presente no próprio módulo de declaração do dado.

Este modelo de programação modular é refinado, dando origem ao modelo de *Programação por Tipos de Dados* (PTD). Através deste método, o programa é analisado e desenvolvido de forma a identificar e caracterizar os dados necessários para seu funcionamento. Para cada dado identificado no problema é criado um tipo de dado que abstrai as características essenciais do dado: os atributos que descrevem os valores que uma variável do tipo pode assumir, mais as funções necessárias para manipular estas variáveis. Torna-se proibida qualquer manipulação de dados sem ser efetuada por uma função declarada para o tipo do dado.

Programação Orientada a Objetos (POO) surgiu como um refinamento do modelo de programação por tipos de dados. Desta vez, além de tratar um problema através da modelagem de seus dados, POO introduz uma forma para descrever similaridades entre dados através do mecanismo de herança.

Estes dois enfoques na atividade de programação, baseados em ações e em dados, são predominantes nas áreas científica e comercial, embora, existam outros modelos, principalmente aqueles relacionados com a área de inteligência artificial:

Programação em Lógica Baseada nas cláusulas de Horn, tem origem na tentativa de automatizar a demonstração de teoremas da lógica de primeira ordem. Deu origem à linguagem Prolog (cuja notação é similar à notação usada em lógica).

Programação Funcional É baseada no conceito de funções recursivas da lógica abstrata e tem como base formal a notação e cálculo lambda. As linguagens que implementam este paradigma, como LISP, não possuem comando de atribuição e todo processamento é implementado pelo cálculo de funções.

Programação com Regras de Produção Surgiu da necessidade de uma linguagem mais específica para a programação de sistemas especialistas em inteligência artificial. Consistem basicamente de um conjunto de regras do tipo

```
se <uma dada condicao e' verificada> entao execute <uma dada acao>
```

que, manipulando uma base de dados global às regras, busca modelar a forma com que o especialista atua de acordo com cada situação possível encontrada no domínio do problema.

2.3 Desenvolvimento Baseado em Ações

Os métodos de desenvolvimento baseados em ações consistem na descrição das ações que o computador deve executar usando, inicialmente uma notação que abstraia o funcionamento global do programa. Esta notação deve ser refinada gradualmente até que cada tarefa resultante corresponda a um comando disponível na linguagem de programação.

Direta ou indiretamente, estas técnicas têm origem no trabalho pioneiro de Dijkstra, em que ele alertava a comunidade cientifica para a incapacidade do ser humano no gerenciamento de projetos de software de grande porte e para a conseqüente necessidade de sistematização e disciplina no desenvolvimento de programas.

2.3.1 Programação Estruturada

O paradigma de Programação Estruturada é o mais antigo que conhecemos e, talvez por ser o primeiro, reflete intrinsecamente a arquitetura interna do computador: a aplicação de operações sobre dados. O termo foi introduzido em 1969 por Dijkstra [Dijks72] ao propor que os comandos

- de atribuição de valores às variáveis;
- de teste condicional (**if-then-else**);
- de iteração (**while-do**);
- de composição de blocos (**begin-end**).

são suficientes para escrever programas. Além disto, a utilização do comando GOTO deve ser evitada. Usando um conjunto reduzido de comandos básicos como estes, a proposta

do modelo de programação estruturada é sistematizar o desenvolvimento de programas através da decomposição modular do programa em programas menores. Um programador usando esta técnica deve tratar um problema da seguinte forma:

> Analisando o problema, divida-o em problemas menores de tal forma que resolvendo cada um destes problemas teremos a solução do problema original.

Na prática, este método propõe um seqüenciamento sistemático na descrição do programa tal que as estruturas das computações sejam refletidas na estrutura do programa. O programa é produzido através de refinamentos sucessivos de tal forma que, a cada passo, uma ou mais instruções do programa em desenvolvimento são decompostas em instruções mais detalhadas. Esta decomposição sucessiva (ou refinamento de especificações) termina quando todas as instruções forem expressas nos termos de uma linguagem de programação subjacente e deve, por conseguinte, ser guiada pelos recursos disponíveis nesta mesma linguagem. Como o resultado da execução de um programa é expresso nos valores atribuídos aos dados e mais dados são introduzidos para realizar o intercâmbio entre as subtarefas ou instruções obtidas, à medida em que tarefas são refinadas, os dados também devem ser refinados, decompostos ou estruturados, e é natural refinar o programa e os dados em paralelo.

2.3.2 Exemplo: Problema das 8 rainhas

O problema: os dados são um tabuleiro de xadrez (com 8 linhas e 8 colunas) e 8 rainhas hostis uma a outra. Encontre uma posição no tabuleiro para cada rainha (uma configuração) tal que nenhuma rainha possa ser tomada por nenhuma outra (ou seja, tal que toda linha, coluna e diagonal do tabuleiro contenha no máximo uma rainha).

Denotando por A o conjunto de todas as configurações obtidas ao posicionar as 8 rainhas no tabuleiro, queremos encontrar uma configuração que satisfaça a uma certa condição p (afirmando que não existem duas rainhas na mesma linha, coluna ou diagonal). Desta forma, uma solução é caracterizada por um x tal que $(x \in A) \wedge p(x)$.

Uma descrição imediata de um tal programa é:

repita
    Gere o próximo elemento de A e chame-o de x
até que $p(x) \vee$ (todos elementos de A foram testados)
se $p(x)$ **então** x é uma solução

Apesar de correto, este algoritmo torna-se bastante ineficiente ao testar todos os 2^{32} elementos de A. Para reduzir este número, eliminando de imediato uma quantidade considerável de elementos que não fornecem uma solução do problema, basta considerar que em cada coluna do tabuleiro deve estar posicionada exatamente uma rainha. Desta forma, pode-se construir um elemento de A ao tentar posicionar uma rainha em cada coluna, utilizando a técnica de *backtracking*.

variáveis
    tabuleiro, ponteiro, seguro;
considerePrimeiraColuna;
repita
    tenteColuna;
    se *seguro*
        então faça
            posicioneRainha;
            considereProximaColuna
            fim-faça
        senão *retorne*
até-que *ultimaColunaFeita* $\vee$ *retornoDaPrimeiraColuna*

Note que este programa é composto por um conjunto de instruções mais primitivas,

- *considerePrimeiraColuna*
- *tenteColuna*
- *seguro*
- *posicioneRainha*
- *considereProximaColuna*
- *retorne*
- *ultimaColunaFeita*
- *retornoDaPrimeiraColuna*

que deverão ser descritas nos passos seguintes. Para prosseguir com o desenvolvimento, que não faremos aqui, basta especificar cada uma destas operações bem como a representação dos dados citados (*tabuleiro*, *ponteiro* e *seguro*).

Este método apresenta as seguintes características:

- A representação de dados obtida durante o desenvolvimento do programa é definida considerando-se principalmente sua conveniência para uma solução eficiente do problema proposto.

- O programa é desenvolvido a partir de seu funcionamento global e este funcionamento é detalhado nos passos seguintes. Chamamos esta técnica de "programação top-down".

- Apresenta uma forma natural para a decomposição do programa em módulos: cada instrução primitiva pode ser isolada em um módulo. No entanto, não impõe nenhum cuidado com relação à dependência de um módulo com relação aos outros.

- Em princípio, cada módulo obtido durante o desenvolvimento é dependente da aplicação específica para a qual foi criado.

2.4 Desenvolvimento Baseado em Dados

Neste modelo de desenvolvimento, a descrição dos sistemas é dirigida pela descrição dos dados necessários para seu funcionamento. O processo de análise do problema é responsável pela definição do comportamento esperado do sistema. Paralelamente a esta atividade, deve-se começar a identificar os dados que serão responsáveis pela condução deste comportamento. O processo de desenvolvimento consiste na implementação do modelo baseado em dados encontrado na análise.

2.4.1 Programação com Tipos de Dados

Antes da criação deste paradigma um tipo de dados era caracterizado apenas pelo conjunto de valores que uma variável pertencente ao tipo pode assumir. Redefinindo este conceito, um tipo de dados passa então a ser definido não somente pelo conjunto de valores mas também pelas funções que manipulam este conjunto (são as *funções de interface* do tipo). Toda e qualquer manipulação efetuada sobre uma variável do tipo deve, obrigatoriamente, ser realizada por funções de interface do tipo. Uma linguagem de programação que ampare este paradigma de programação deve acusar erro na tentativa de se manipular um dado através de funções ou operadores não definidos para este fim.

Baseado nesta nova definição de tipos de dados surge o paradigma de *Programação por Tipos de Dados* (PTD). O processo de programação consiste, a partir de agora, em descrever tipos de dados: englobar, dentro de módulos, descrição do conjunto de valores com todas as suas funções de interface. Temos então o paradigma:

Decida que tipos você necessita; para cada tipo, apresente um conjunto de funções definidas exclusivamente para sua manipulação.

Por exemplo, se no desenvolvimento de um programa for necessária uma estrutura de dados para manipular pilhas de números inteiros, basta criar um novo tipo de dados, digamos PilhaDeInteiros, que implemente este tipo:

- atributos para representar pilhas:
  - um vetor de inteiros;
  - um marcador da posição de topo da pilha;
- rotinas para
  - criar uma pilha (inicialmente vazia);
  - empilhar um elemento;
  - desempilhar um elemento;
  - consultar se a pilha está vazia;
  - consultar o valor do elemento de seu topo.

A característica fundamental deste paradigma é a separação dos dados (junto com suas funções de interface) em módulos; desta forma eles estão "escondidos" dos usuários do tipo. Aquele que utilizar o tipo PilhaDeInteiros em seu programa não precisa se preocupar como ele foi implementado; precisa apenas saber:

- que a pilha manipula qualquer número inteiro corretamente
- se existe alguma restrição com relação ao número máximo de elementos empilhados (e o que acontece se for extrapolado este máximo)
- que funções tem a disposição para manipular estas pilhas

Na eventual ocorrência de algum erro na manipulação de pilhas, o responsável será, certamente, o sujeito que implementou este tipo (se alguma coisa deu errado, foi porque a pilha foi manipulada de forma equivocada). Desta forma fica bastante facilitada a localização e correção de erros (uma vez que os tipos de dados devem estar coerentemente separados em módulos).

Por outro lado, se for necessária alguma modificação na representação das pilhas (por exemplo, mudar de representação estática do vetor para uma dinâmica), não serão necessárias modificações nos programas que utilizam esta pilha (desde que não seja modificada também a interface do tipo). Assim torna-se também facilitada a manutenção de programas.

Note que para a implementação das funções de interface de um tipo de dados será necessário explicitar as ações que o computador deve executar. Nesta fase também devemos ser metódicos e sistemáticos; para isto, programação estruturada é uma boa solução. Programação estruturada auxilia na descrição da seqüência de ações que o computador deve enquanto PTD auxilia na decomposição do problema pela modelagem de dados.

2.4.2 Exemplo: Contagem das folhas de uma árvore

Para ilustrar o método de desenvolvimento baseado em dados, usaremos um exemplo de autoria de G. Booch [Booch82] descrevendo uma versão primitiva de seu método para análise e desenvolvimento orientado a objetos.

Para isto será proposto um problema que será resolvido (aqui não faremos isto completamente) segundo os passos que o autor considera relevantes para a solução do problema. O problema proposto a ser resolvido é o de contar o número de folhas de uma árvore binária.

Definir o problema

Uma árvore binária é uma estrutura de dados na qual cada nó tem duas ramificações ou nenhuma. Uma árvore é uma folha isolada ou consiste de duas subárvores. Se uma *árvore* é uma folha isolada então $folhas(árvore) = 1$ e se consiste de duas subárvores,

$$folhas(árvore) = folhas(árvore_1) + folhas(árvore_2)$$

(onde $árvore_1$ e $árvore_2$ denotam as subárvores a direita e a esquerda, respectivamente).

Nosso trabalho é desenvolver um sistema que conte o número de folhas de uma dada árvore.

Desenvolver uma estratégia informal para o mundo abstrato

Dentre as várias formas de se contar o número de folhas de uma árvore, usaremos um algoritmo que parece bastante intuitivo; tal abordagem deve refletir diretamente nossa visão abstrata do mundo. Assumiremos somente que dispomos de três estruturas básicas de controle (seqüenciamento, condicional e iteração) e recursos para definir operações e objetos para nosso mundo abstrato (extensibilidade). Fixadas estas restrições, apresentaremos nossa estratégia informal:

Manteremos uma pilha contendo as partes da árvore que ainda não foram contadas. Inicialmente, empilharemos a árvore a ter suas folhas contadas na pilha vazia e atribuímos o valor inicial zero ao contador de folhas desta árvore. Enquanto a pilha não estiver vazia, desempilharemos uma árvore e a examinaremos; se a árvore consistir de uma folha isolada, incrementaremos o contador de folhas e descartamos a árvore; se a árvore apresentar duas subárvores, empilharemos sua subárvores direita e esquerda. Quando a pilha estiver vazia exibiremos como resposta o valor do contador de folhas.

Formalizar a estratégia

O passo seguinte é refinar nossa visão do mundo abstrato, até obter a implementação do programa. Este passo compreende as seguintes atividades:

- Identificar objetos e seus atributos
- Identificar as operações sobre os objetos
- Estabelecer a interface
- Implementar as operações

Identificar objetos e seus atributos Para isto, basta analisar o comportamento esperado do sistema e identificar os dados manipulados (indicados em negrito):

Manteremos uma **pilha** contendo as **partes da árvore** que ainda não foram contadas. Inicialmente, empilharemos a **árvore** a ter suas folhas contadas na **pilha vazia** e atribuímos o valor inicial zero ao **contador de folhas**. Enquanto a **pilha** não estiver vazia, desempilharemos uma **árvore** e a examinaremos; se **ela** consistir de uma **folha isolada**, incrementaremos o **contador de folhas** e descartamos a **árvore**; se a **árvore** apresentar duas **subárvores**, empilharemos suas **subárvores direita e esquerda**. Quando a **pilha** estiver vazia exibiremos como resposta o valor do **contador de folhas**.

Desta avaliação, podemos constatar que nossos tipos básicos são

- *contadorDeFolhas*
- *pilha*
- *subarvoreDireita*
- *subarvoreEsquerda*
- *arvore*

Neste ponto do desenvolvimento, nomeamos os objetos que são de interesse para a aplicação e estamos prontos para enumerar as operações que caracterizam o comportamento de cada um destes objetos.

Identificar as operações sobre os objetos Novamente, analisamos o comportamento desejado do sistema e identificamos as operações efetuadas sobre os dados (indicadas em negrito):

Manteremos uma pilha contendo as partes da árvore que ainda não foram contadas. **Inicialmente, empilharemos** a árvore a ter suas folhas contadas na pilha vazia e **atribuímos o valor inicial zero** ao contador de folhas desta árvore. Enquanto a pilha não **estiver vazia, desempilharemos** uma árvore e a **examinaremos**; se **a árvore consistir de uma folha isolada, incrementaremos** o contador de folhas e **descartamos** a árvore; se **a árvore apresentar duas subárvores, empilharemos** suas subárvores direita e esquerda. Quando a pilha **estiver vazia, exibiremos** como resposta o valor do contador de folhas.

Assim, as operações necessárias para manipular os dados são:

- Para *contadorDeFolhas*

  Exibe apresenta o valor do contador

  Incrementa soma um ao valor do contador

  Zera atribui o valor zero ao contador

- Para *pilha*

  Vazia consulta se existe algum elemento empilhado

  Empilha empilha um elemento

  EmpilhaInicial empilha o primeiro elemento (note que esta função corresponde a empilhar um elemento em uma pilha inicialmente vazia)

  Desempilha desempilha um elemento

- Para *subarvoreDireita*, *subarvoreEsquerda* e *arvore*

  Leitura lê uma árvore

  ENoIsolado verifica se uma árvore corresponde a um nó isolado

  Quebra decompõe uma árvore, retornando suas subárvores direita e esquerda

  Descarta destrói uma árvore

Estabelecer a interface Neste ponto, começamos nosso desenvolvimento pela descrição dos dados obtidos como instâncias de tipos abstratos de dados. Completamos nosso primeiro nível de desenvolvimento com a descrição dos relacionamentos entre estes objetos.

Implementar as operações Para não estender demasiadamente este desenvolvimento, apresentaremos somente o módulo principal do programa (usando uma pseudolinguagem):

```
inclua modulo descrevendo Pilha
inclua modulo descrevendo Arvore
inclua modulo descrevendo Contador

variaveis
   contadorDeFolhas: Contador;
   arvore: Arvore;
   pilhaDeArvores: Pilha;

inicio
   Leia(arvore);
   EmpilhaInicial(pilhaDeArvores, arvore);
   Zera(contadorDeFolhas);
   enquanto nao Vazia(pilhaDeArvores)
        faca
           Desempilha(pilhaDeArvores, arvore);
           se ENoIsolado(arvore)
                entao
                   Incrementa(contadorDeFolhas);
                   Descarta(arvore);
                fim-entao
                senao
                   Quebra(arvore, arvoreDireita, arvoreEsquerda);
                   Empilha(arvoreDireita);
                   Empilha(arvoreEsquerda);
                fim-senao
        fim-faca
   Exibe(contadorDeFolhas);
fim
```

Comentários sobre o programa obtido:

- O programa é construído através da análise de seu comportamento desejado e da distribuição das atividades (que implementam este comportamento) entre os dados manipulados pelo programa.

- Apresenta uma forma natural para a decomposição dos programas em módulos: faça de cada tipo um módulo.
- É recomendável implementar estes tipos obtidos independentemente do domínio deste problema de tal forma que seja possível reaproveitá-los em outros problemas. O tipo Contador, por exemplo, pode ser utilizado para contar carneirinhos também.

2.5 Ciclo de Vida de um Programa

As idéias primitivas sobre o ciclo de vida de um programa surgiram na década de 60 ao se tentar descrever as fases envolvidas na atividade de produção de um software. O modelo mais antigo, chamado de *waterfall* (cuja tradução literal forneceria algo como "cascata"), descreve a produção de um software como uma seqüência linear de fases (como elaboração, análise, desenvolvimento, implementação, etc.) tais que cada fase é iniciada com os produtos resultantes da fase anterior e fornece recursos para a seguinte (veja figura 2.2). Este modelo propõe que a produção de um programa é conseqüência da realização bem sucedida de cada uma destas atividades enquanto que o fracasso nas atividades envolvidas em uma das fases implica na reformulação da fase anterior.

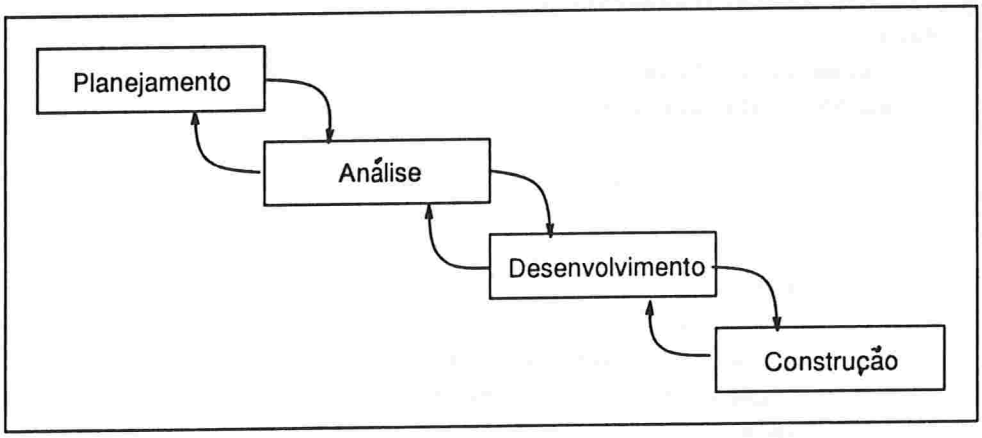

Figura 2.2: Modelo waterfall de ciclo de vida

O conceito de ciclo de vida proporcionou embasamento para uma análise mais completa de sistemas de software. Até o inicio dos anos 70 os projetos computacionais eram organizados de forma a minimizar os custos da fase de desenvolvimento em vez do custo de todo o ciclo de vida.

Por exemplo, nesta época, a análise de três grandes projetos militares de software, estimou que tais sistemas tipicamente têm um ciclo de vida de 16 anos, consistindo de

um estágio de formulação conceitual e especificação de necessidades de 6 anos, um estágio de desenvolvimento de 2 anos e um estágio de operação e manutenção de 8 anos. Este resultado mostrava que a fase de desenvolvimento, que até então se acreditava que era responsável pela maior parte do custo de um software, correspondia a apenas um oitavo do ciclo de vida.

O ciclo de vida de um software, como formalizado pelas agências do departamento de defesa do governo norte-americano, consiste de:

- Um estágio de formulação conceitual e especificação de necessidades;
- Um estágio de desenvolvimento do software;
- Um estágio de operação e manutenção.

Por sua vez, estas fases podem ser refinadas de tal forma que que a fase de desenvolvimento consista de:

- Um estágio de análise de necessidades;
- Um estágio de projeto do programa;
- Um estágio de implementação e depuração de erros;
- Um estágio de testes e avaliação.

A partir de então começaram a ser buscadas técnicas para o desenvolvimento de sistemas que privilegiassem todas as fases do ciclo de vida, de forma a minimizar o custo total dos sistemas produzidos.

Um modelo de ciclo de vida mais recente é o modelo de *espiral* (veja figura 2.3) que considera a produção de um software como uma atividade cíclica, tal que após o término de uma versão de um sistema volta-se ao planejamento para uma nova, e inclui novas atividades como a criação de protótipos do sistema e sua manutenção.

A importância dos modelos de ciclo de vida no desenvolvimento de programas reside no poder que estes modelos têm de descrever todos os processos envolvidos na produção de um software. Assim, uma forma de avaliar a qualidade de um método de programação é analisar a forma com que ele administra o ciclo de vida dos sistemas.

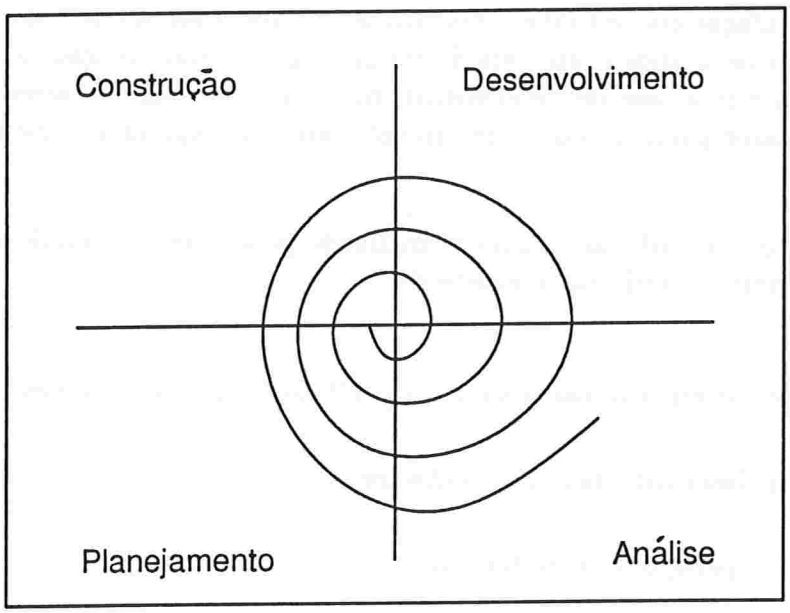

Figura 2.3: Modelo espiral de ciclo de vida

2.6 Suporte ao Desenvolvimento de Programas

Nestes últimos anos, principalmente com a almejada meta de produção de software em larga escala, tem-se buscado formas para aumentar a produtividade no projeto e desenvolvimento de sistemas. Para isto, são pesquisadas formas eficazes para dirigir o todo processo de produção de um software, sejam elas através da imposição de roteiros que induzam à disciplina ou formas para amparar o programador na codificação e testes dos programas.

Dentre as propostas que tem surgido para amparar o desenvolvimento de sistemas, destacamos:

Novas linguagens de programação: fornecendo maior poder de descrição de modelos reais e recursos para uma melhor administração de todo o ciclo de vida do sistema.

Ambientes de desenvolvimento de sistemas: que facilitam a atividade de programação fornecendo recursos para edição e depuração de erros de programas. Os ambientes baseados em janelas facilitam a programação modular com a possibilidade de se editar vários módulos ao mesmo tempo.

Notações gráficas: que facilitam a descrição do modelo conceitual implementado pelo programa e auxiliam no retratamento das partes e atividades mais importantes do sistema.

Ferramentas CASE: tratam-se de ferramentas computacionais que amparam a equipe de desenvolvimento no exercício das atividades envolvidas na produção de um software (segundo um método específico de desenvolvimento de sistemas).

Reutilização de software: a idéia de reutilização de software consiste na tentativa de se reaproveitar partes de sistemas já implementados para produzir novos sistemas.

Geradores de aplicações: são programas que aceitam como entrada a descrição de uma aplicação computacional e produzem um programa que implementa esta aplicação desejada.

2.7 Referências Bibliográficas

Nas partes iniciais deste capítulo foi consultado [Wegner76], que relata os primeiros 25 anos das linguagens de programação, como literatura básica de referência. Além deste, foram usados [Booch91, Mona92, Stro88] como textos de apoio para a caracterização de métodos de desenvolvimento de programas.

Programação estruturada foi consultada na obra-prima de Dijkstra [Dijks72] e o exemplo das 8 rainhas foi consultado em [Wirth71]. Para ilustrar o método de programação baseado em dados, tomaremos o exemplo apresentado em [Booch82]. Embora este artigo se apresente como DOO, é importante relatar que seu conteúdo é voltado para a linguagem Ada (que não deve ser considerada uma linguagem OO por não apresentar herança).

O conceito de ciclo de vida de um software foi pesquisado em [Wegner76] enquanto que os modelos de ciclo de vida waterfall e espiral, embora apresentados em versões adaptadas, foram tomados de [Taka90].

Capítulo 3

Falando de Objetos

Desde sua criação, POO tem proporcionado mudanças, muitas vezes sensíveis, na forma de abordar o problema de elaboração de programas de computadores. Além de introduzir uma terminologia particular POO mudou o enfoque nas atividades de análise e desenvolvimento de sistemas. Neste capítulo abordaremos as principais características das linguagens orientadas a objetos dando ênfase nos recursos que elas oferecem para facilitar o desenvolvimento de programas.

3.1 Histórico do Modelo de POO

Apesar de seu reconhecimento e sucesso recentes, o modelo de Programação Orientada a Objetos conta quase 30 anos de existência (desde seu embrião com origem na linguagem Simula).

Este paradigma surgiu na década de 60 com o desenvolvimento da linguagem Simula. Criada na Noruega para facilitar a descrição formal de sistemas de simulação, esta linguagem consistia em uma extensão ao Algol 60. Aperfeiçoada, deu origem à Simula 67, na qual são introduzidos conceitos inovadores como objeto e classe que se tornariam essenciais na caracterização de linguagens OO.

No entanto, foi somente, na década de 80, com a divulgação da linguagem Smalltalk, que POO se impôs como um modelo de computação alternativo a programação estruturada. O projeto Smalltalk foi iniciado em 1969 com a criação do Grupo de Pesquisa de Aprendizagem no Centro de Pesquisas de Palo Alto da Xerox. Com este grupo, Alan Kay procurou realizar sua "visão" de um mundo em que pudéssemos dispor de sistemas "amigáveis" de interface com computadores. Para implementar suas idéias, um computador deveria fornecer um monitor de vídeo com uma boa resolução gráfica e um ambiente

de programação e operação baseado em janelas no qual qualquer serviço fosse disponível a um toque de tecla.

Durante este projeto surgiu a necessidade de uma linguagem acoplada ao ambiente, motivando a criação de uma nova linguagem, também chamada Smalltalk, que, depois de algumas versões iniciais (em 1972, 1974 e 1976), no início da década de 80, está finalmente pronta, implementada e disponível com seu sistema baseado num ambiente com sobreposição de janelas. Apresentada ao público com grande destaque na revista Byte em agosto de 1981 [Byte81], Smalltalk divulgava quase todos os conceitos e terminologias daquilo que a partir de então passou a ser conhecido como Programação Orientada a Objetos. O sistema Smalltalk implementava as visões de Alan Kay e de brinde trazia a nova linguagem. Além de introduzir os principais conceitos de POO esta linguagem apresentava como características:

- interface amigável com o usuário: um ambiente baseado em janelas e menus de opções de serviços;
- ambiente de programação: recursos para edição, execução e depuração de erros de programas escritos em Smalltalk;
- sintaxe (estruturalmente) próxima da linguagem humana;
- cada programa escrito é incorporado ao sistema, ficando disponível para utilização em outros programas.

Apesar do sucesso de seus conceitos, Smalltalk não ganhou muitos adeptos. Alguns motivos determinantes para isto foram:

- baixo desempenho, principalmente por tratar-se de uma linguagem interpretada;
- necessidade do "ambiente Smalltalk" para executar programas escritos em Smalltalk;
- rigor conceitual: em Smalltalk, tudo é objeto (até mesmo as operações básicas do computador devem ser realizadas através da comunicação entre objetos) e isto dificultava sua introdução a pessoas acostumadas com outros modelos de computação.

Esses motivos, aliados com as qualidades do modelo baseado em objetos, determinaram a criação de linguagens híbridas, juntando idéias de POO a outros paradigmas. Reflexo disto é o surgimento recente de extensões a C, Pascal, LISP, Prolog (dentre tantas) com a inclusão de objetos.

Além de propiciar o surgimento de novas linguagens e a adaptação de antigas, este modelo de programação passou ser utilizado não somente como um modelo de linguagens,

mas também como uma técnica de análise e desenvolvimento de sistemas. A partir destes trabalhos voltados para a utilização de idéias de POO em análise e desenvolvimento, foram difundidos os termos Análise Orientada a Objetos (AOO) e Desenvolvimento Orientado a Objetos (DOO).

3.2 Conceito de POO

Definir precisamente o significado do termo "programação orientada a objetos" é uma tarefa difícil (para não dizer impossível) uma vez que não existe consenso entre as personalidades da área. Depois de relatar inconsistências nas caracterizações de POO, Nierstrasz diz: "Uma definição completa do que significa ser orientado a objetos não é, conseqüentemente possível, embora possamos julgar quando um sistema ou linguagem é "mais" orientado a objetos que outros"[Nier89]. Por causa desta falta de consenso, são encontradas diversas caracterizações de POO:

em smalltalk Com Smalltalk, o modelo de POO foi caracterizado através de conceitos como objeto, classe, método, herança e mensagem.

Sua característica fundamental é a uniformidade conceitual: tudo é objeto e todos os objetos atuam somente enviando e recebendo mensagens; até mesmo as operações mais elementares do computador devem ser efetuadas por intermédio de troca de mensagens entre objetos.

Por exemplo, se quisermos calcular $7 + 8$ (que em C++ é calculado diretamente, somando os inteiros 7 e 8) em Smalltalk será necessário "enviar a mensagem" $7 + 8$; como o método + está definido para a classe do objeto 7 e este método aceita um objeto inteiro como parâmetro, calcula-se a soma, gerando o objeto 15. Este "radicalismo" faz com que os seguidores deste modelo considerem que C++ não é uma linguagem orientada a objetos.

em "simulação" A primeira caracterização de POO é originária da linguagem Simula e hoje também defendida pelos adeptos da linguagem Beta. Baseado nos problemas de simulação e descrição de sistemas para os quais Simula foi criada, este grupo vê POO como uma "filosofia" de modelagem do mundo real:

> Com Programação Orientada a Objetos, a execução de um programa é considerada como um modelo físico, simulando o comportamento de uma parte real ou imaginária do mundo.

A partir desta definição, passam a definir o conceito de modelo físico, além de muitos outros que surgem durante a explicação. Não entraremos em detalhes sobre esta abordagem (consulte [Nyga86]).

a partir de conceitos de objetos A abordagem mais freqüente na apresentação de POO é enumerar algumas de suas características e justificar que estas são as principais apresentadas por uma linguagem ou sistema OO. Dentre elas, as mais citadas são encapsulamento e herança; outros termos bastante utilizados são

- programação por tipos de dados;
- acoplamento dinâmico;
- polimorfismo;
- dados ocultos.

O que existe em comum em todas estas definições é o conceito de objeto como um tipo de dados (um pacote contendo a descrição de um tipo de dado e as funções necessárias para a manipulação de instâncias deste tipo) e herança para definir hierarquia de classes.

Programação por Tipos de Dados e Herança No entanto, a forma mais simples de definir POO e caracterizá-la como um modelo de programação baseado na descrição de tipos de dados no qual usamos hierarquia de tipos (herança) para modelar similaridades entre tipos.

3.2.1 Conceito de Objeto

O conceito de objeto está intimamente ligado ao conceito de tipo de dados no paradigma de PTD: um objeto é definido por um conjunto de atributos e funções que manipulam estes atributos:

- o conjunto de atributos descreve a representação de um elemento a ser modelado, ou seja, os atributos que caracterizam tal elemento.
- as funções descrevem as únicas operações permitidas para manipular este elemento; são as ações que caracterizam o comportamento deste elemento.

Freqüentemente, o termo objeto é utilizado com ambigüidade para nos referirmos tanto ao tipo de dado (chamado de *classe*) como a uma variável pertencente a um tipo de dado (chamado de *instância*).

Para exemplificar, apresentamos, a seguir, a declaração de uma classe, codificada em C++, para implementar pilhas de números inteiros:

```
class Pilha{
    int *vetor,     // armazenamento de elementos empilhados
        topo,       // indice da posicao do topo da pilha
        tamanho;    // tamanho da pilha
public:
    Pilha(int tam= 100);
    ~Pilha();
    char empilha(int elem);
    int  desempilha();
    char vazia();
    int  topo();
};
```

Os atributos e as funções declarados na classe são chamados de *membros* da classe. As funções são chamadas especificamente de *funções membro*. Os membros que aparecem após o rótulo public: são os membros públicos da classe Pilha, são eles que estabelecem a interface entre os usuários da classe com as instâncias que manipulam.

Note que a classe Pilha listada acima apenas declara a representação e as funções disponíveis para a manipulação de tal objeto. Falta, ainda, fornecer a implementação de cada uma de suas funções membro:

```
Pilha::Pilha(int tam= 100)
   {  vetor= new int[tam];
      topo= 0;
      tamanho= 100;
   }
Pilha::~Pilha()
   {  delete vetor; }
char Pilha::char empilha(int elem)
   {  if (topo ==tamanho)
         return 0;
      else { vetor[topo++]= elem; return 1; }
   }
int Pilha::int  desempilha()
   {  assert (topo);
      return vetor[--topo];
   }
char Pilha::char vazia()
   {  return !topo; }
int Pilha::int  topo()
   {  assert (topo);
      return vetor[topo-1];
   }
```

3.2.2 Conceito de Linguagem Orientada a Objetos

Uma linguagem é considerada *Linguagem Orientada a Objetos* se ela dá suporte ao modelo de POO. Devido à falta de uma definição precisa do significado de POO, ocorre também certa imprecisão na caracterização de linguagens OO. [Wegner89] usa conceitos como objeto, classe e herança para classificar linguagens que manipulam objetos em:

linguagens baseadas em objetos são aquelas que amparam manipulação de objetos, ou seja, permitem a descrição de dados junto com as operações com permissão para manipulá-los.

linguagens baseadas em classe são aquelas não só amparam a manipulação de objetos, mas nas quais estes objetos pertencem a classes.

linguagens orientadas a objetos são linguagens baseadas em classes e permitem herança entre elas.

Assim, segundo esta definição, uma linguagem para ser considerada orientada a objetos deve amparar o modelo de PTD com a adição do mecanismo de herança entre tipos herança. Em suma, ela deve:

- amparar o programador na definição de seus tipos de dados (classes) de tal forma que cada classe funcione exatamente como uma forma para a declaração de instâncias caracterizadas pelos atributos e funções declaradas na própria classe;
- estas classes devem se comportar como tipos pré-definidos da própria linguagem, ou seja, deve ser possível definir ponteiros vetores, parâmetros de funções, etc., usando esta classe;
- permitir a especialização do comportamento de uma classe através da criação de uma classe derivada.

3.3 Propriedades das Linguagens OO

Segundo Stroustrup [Stro88], existe uma sensível diferença entre uma linguagem que permite um estilo de programação e aquela que ampara este estilo. No primeiro caso, a linguagem simplesmente "não impede" que o programador use o estilo de programação, sem oferecer recursos apropriados para sua correta utilização; o programador deverá exercer esforço ou perícia excepcionais para utilizar o estilo. Uma linguagem que ampara um

estilo de programação se oferece recursos que tornam conveniente (fácil, seguro e eficiente) a utilização do estilo.

A seguir, estudaremos as características normalmente citadas com o paradigma de objetos, aquelas principais que as linguagens que amparem objetos devem oferecer. São elas:

- encapsulamento
- modularidade
- abstração
- herança
- polimorfismo

3.3.1 Abstração

Para introduzir o conceito de abstração, considere o seguinte exemplo: ao modelar um sistema para controle de empréstimos de uma biblioteca, certamente enfrentaremos o desafio de modelar o conceito de usuário da biblioteca. Modelar este conceito equivale a responder às seguintes perguntas:

- Que características são relevantes para se descrever tais usuários?
- Que comportamento tais usuários apresentam dentro do sistema?

Certamente características como cor dos olhos e corte de cabelo são irrelevantes para o funcionamento do sistema. Da mesma forma, acredita-se que todos os usuários costumam ingerir alimentos, embora também este comportamento seja desnecessário para caracterizar tais usuários (embora, até hoje, eu mesmo nunca tenha encontrado um usuário de biblioteca que não ingira alimentos). O essencial, especificamente para o sistema de controle de empréstimos de uma biblioteca, é descrever que

- todo usuário tem uma identificação (nome, endereço, telefone, etc.)
- todo usuário tem um cadastro na biblioteca que, além de sua identificação, apresenta sua situação com a biblioteca (descrição dos livros que tem emprestado)

- a única atividade que tais usuários apresentam que influencia o comportamento da biblioteca é capacidade que estes têm para solicitar o empréstimo de livros do acervo (eventualmente podem fazer a reserva de um título em circulação ou encomendar a compra pela biblioteca de um novo título)

Independentemente de uma linguagem de programação, dizemos que "a abstração usuário" tem como atributos (dentre tantos atributos importantes para a caracterização de usuários de biblioteca)

- nome;
- endereço;
- telefone;
- ficha apresentando sua situação de empréstimos.

e que cada uma destas abstrações de usuário podem realizar as seguintes operações:

- solicitar o empréstimo de livros;
- devolver um livro que tomou emprestado;
- solicitar a reserva de livros.

Abstração é caracterizada como a descrição das propriedades essenciais de um conceito. No exemplo anterior, foram abstraídas as características principais do "conceito" de usuário de biblioteca. Booch, por sua vez, define abstração como "uma abstração denota as características essenciais de um objeto que o distingue de todos os outros tipos de objetos e assim provê fronteiras conceituais definidas nitidamente, relativas à perspectiva do observador" [Booch91]. Na fase de análise de seu método , Booch usa o termo abstração no lugar de classe para designar o modelo de um conceito; classe é usado no desenvolvimento apenas como uma ferramenta (da linguagem de programação) para descrever uma abstração.

Uma abstração focaliza-se na visão externa de um objeto e serve para separar o comportamento essencial de um objeto de sua implementação. Note que a noção de observador para uma abstração é fundamental. Por exemplo, para o sistema de controle de biblioteca, a cor dos olhos e o corte de cabelo são irrelevantes; tomando-se, desta vez, como observador uma bibliotecária a procura de marido, esta informação pode se tornar essencial.

Em POO, identificar uma abstração é o primeiro passo para a criação de uma classe. Por exemplo, uma implementação (preliminar) da abstração usuário é:

```
class Usuario{
    char  *nome, *telefone, *endereco;
    Livro *retirados[10];
public:
    Usuario();
    Usuario(char*, char*, char*);
    ~Usuario();
    int atribuiNome(char*);
    int atribuiEndereco(char*);
    int atribuiTelefone(char*);
    char *retornaNome() const;
    char *retornaEndereco() const;
    char *retornaTelefone() const;
    int emprestaLivro(Livro*);
    int devolveLivro(Livro*);
};
```

Uma propriedade fundamental das linguagens orientadas a objetos é:

> Na POO, cada classe implementa uma abstração.

Abstração é uma palavra chave em POO por ser o mecanismo que dirige a caracterização das classes que compõem um sistema OO. Este é, sem dúvida, o grande desafio àqueles que trocam de uma linguagem procedimental para uma linguagem OO: deixar de dirigir a programação pela descrição das atividades do computador e passar a descrever abstrações que ocorrem no domínio do problema. Nesta atividade reside aquilo que é normalmente comentado por "pensar em objetos". Finalizando,

> Na POO toda classe implementa uma abstração de um elemento do mundo real ou uma componente necessária para construção de um modelo conceitual para o problema.

3.3.2 Modularidade

O conceito de modularidade corresponde à divisão de um programa em partes. Enquanto que qualquer linguagem que ofereça recursos para criação de sub-rotinas seja suficiente para se escrever programas modulares, foi com programação estruturada que este recurso ganhou importância no processo de desenvolvimento de sistemas. Mais do que particionar o código do programa, programação estruturada fornecia um método para decompor o

problema em problemas menores de tal forma que cada subproblema corresponda a uma sub-rotina.

Dizemos que um programa é *modular* se está dividido em partes (os *módulos*) de forma a facilitar seu gerenciamento. Com relação às linguagens, dizemos que uma linguagem ampara programação modular se ela oferece recursos para facilitar a divisão de programas em módulos.

Idealmente, os módulos de um programa devem apresentar as seguintes características:

- devem ser independentes. Alterações efetuadas em um módulo não devem implicar em alterações nos demais módulos do programa. Além disso, cada módulo deve ser capaz de ser testado e compilado separadamente.
- devem ser relativamente pequenos: cada módulo deve apresentar um grau de complexidade suficientemente pequeno para ser gerenciado por uma única pessoa;
- não devem afetar o desempenho do programa: a divisão em módulos não deve acrescentar tempo considerável na execução do programa.

Enquanto que programação estruturada combatia o uso do goto, programação modular combate o uso de variáveis globais (dados que são indiscriminadamente manipulados por qualquer pessoa). Na ocorrência de erros no comportamento destes dados, ninguém sabe de quem é a culpa. Dados, a partir de agora, devem ser protegidos, manipulados por um conjunto restrito de rotinas incluídas no módulo de declaração do dado (e livre da ação de "estranhos").

Em POO, a modelagem de dados oferece uma nova abordagem para a realização de programação modular no processo de programação. Neste paradigma, cada módulo implementa uma abstração identificada na análise ou desenvolvimento do problema. Nas linguagens orientadas a objetos, módulos funcionam como unidades de descrição e implementação de classes. Por exemplo, em C++ estamos usando modularidade quando escrevemos a linha

```
#include"pilha.h"
```

em nossos programas (supondo que a classe Pilha que implementamos na página 29 está armazenada no arquivo pilha.h). Desta forma, um programa escrito com base no modelo de POO, pode ser visto como um módulo principal que utiliza dados descritos em classes (que são implementadas em módulos separados).

3.3.3 Encapsulamento

Pode-se dizer que encapsulamento é uma conseqüência imediata da caracterização de objeto: cada classe consiste em um "pacote" contendo a representação e o comportamento de um tipo de dados. Na prática, encapsulamento define a forma na qual foi implementada a programação modular nas linguagens OO: cada módulo apresenta a descrição de uma classe individual. O código do programa é distribuído entre os objetos (cada objeto apresenta a parte do código estritamente necessária para sua manipulação). Desta forma, os usuários de uma classe são separados da implementação da própria classe:

- o usuário não tem acesso sobre a representação interna dos objetos que usa;
- para manipular estes objetos, ele pode usar "apenas" as operações que foram definidas (dentro da própria classe) para sua manipulação.

Encapsulamento denota esta característica de se "esconder" a representação dos dados dos usuários da classes e fornecer um conjunto de funções para estabelecer a interface destes dados com os usuários. Por exemplo, abaixo está apresentada a parte de declaração de uma classe para abstrair informações pessoais (no exemplo, apenas o nome, o endereço e o telefone da pessoa).

```
#include "Endereco.h"
class Pessoa{
    char    *nome, *telefone;
    Endereco residencia;
public:
    Pessoa();
    Pessoa(char*, char*, char*);
    ~Pessoa();
    atribuiNome(char*);
    atribuiEndereco(const Endereco &);
    atribuiTelefone(char*);
    char *retornaNome() const;
    Endereco& retornaEndereco() const;
    char *retornaTelefone() const;
    void imprime() const;
};
```

Para manipular um objeto desta classe basta incluir o módulo com a declaração do tipo (que supomos que seja chamado pessoa.h) e já será possível declarar instâncias da classe Pessoa:

```
#include"pessoa.h"
void main()
{
    pessoa umaPessoa;   // umaPessoa e' uma instancia da classe pessoa
    // ...
    umaPessoa.imprime();
}
```

Se no meio deste programa for escrito o comando

```
strcpy(umaPessoa.nome, "Paranguaricutiriniquaro");
```

deverá ser acusado um erro de compilação devido à tentativa de manipular um membro do objeto **umaPessoa**, **nome**, que não tem permissão para ser manipulado pelos usuários da classe. Para efetuar esta operação, o usuário da classe deverá procurar uma função de interface da classe que a implemente.

Realmente, os usuários de uma classe não necessitam nem mesmo saber como ela foi implementada. Devem apenas ter alguma garantia de que a classe está correta (segundo suas especificações), além de saber quais são as funções que são disponíveis para que manipulem objetos desta classe.

Note ainda que muitas das funções apresentadas para a classe **Pessoa**, como a função **retornaNome()**, são tão simples que a manipulação direta do atributo tornaria o programa mais eficiente. Mas é justamente esta a filosofia de POO: o que importa em POO é manter os atributos que qualificam um objeto longe das manipulações alheias. Por este motivo é importante, sempre que criar uma nova classe, fornecer um conjunto apropriado de funções de interface para que seus usuários consigam manipular o objeto sem grandes problemas.

Um detalhe importante nesta classe é que um de seus atributos, **residencia**, apresenta um tipo "esquisito" (no caso, **Endereco**). Na verdade, está especificado que este atributo é um objeto da classe **Endereco** que, acreditamos e torcemos, deve manipular o conceito de endereço. Desta forma podemos particionar a descrição de um objeto com a criação de novos objetos para descrever seus atributos. E o mais importante é que não precisamos também saber como a classe **Endereco** foi implementada (desde que, é claro, ela implemente corretamente uma abstração do conceito de endereço). Necessitaremos apenas de saber quais funções temos à disposição para a manipulação deste atributo.

É importante enfatizar que embora a declaração desta classe forneça detalhes sobre sua interface, ela não deve ser usada como um texto de documentação da própria. Para isto é recomendável escrever um texto específico para esta documentação, descrevendo, dentre

tantas informações úteis, como é modelado o objeto, quais são as funções de interface e eventuais restrições consideradas na modelagem conceitual.

3.3.4 Herança

Herança é uma das características fundamentais de POO que consiste no mecanismo para criação de novas classes ao aproveitar (*herdar*) características de classes previamente existentes. Chamamos de *classe base* da derivação (ou *superclasse*) a classe cujas características foram usadas para criar uma classe herdeira; por sua vez, a classe herdeira é chamada de *classe derivada* (ou *subclasse*). Conceitualmente, ao criar uma classe derivada manifestamos que esta classe define um objeto que pode ser visto como pertencente à classe base mas que possui características específicas que os objetos da classe base não apresentam.

Por exemplo, a partir da classe Pessoa apresentada na página 35 podemos criar classes derivadas como Cliente, Fornecedor e Funcionario, especificando que, embora as instâncias destas novas classes apresentem as características e o comportamento de pessoas, apresentam também características específicas (veja figura 3.1). A classe Funcionario, por exemplo, pode apresentar atributos como funcaoExercida e salario e funções como aumentaSalario(), como listado a seguir:

```
class Funcionario: public Pessoa {
   float  salario;
   Funcao aFuncaoExercida;
public:
   // ...
   void aumentaSalario(float percentual);
   // ...
};
```

Desta forma, um objeto da classe Funcionario apresenta os atributos nome, telefone e residencia, herdados de Pessoa, mais os atributos salario e aFuncaoExercida, específicos para objetos da classe derivada. Esta classe apresenta também as funções membro da classe base (como retornaNome() e atribuiEndereco())[1] e as funções declaradas na própria classe derivada (como aumentaSalario(float)).

Um detalhe importante é que mesmo uma classe derivada pode servir de classe base para uma nova derivação, como uma classe ClienteEspecial derivada de Cliente.

As modificações (com relação à classe base) que podem ser realizadas na classe derivada depende de cada linguagem. Em Smalltalk cada classe derivada pode:

[1] No caso desta declaração de classe derivada em C++, isto ocorre porque a derivação é *pública*.

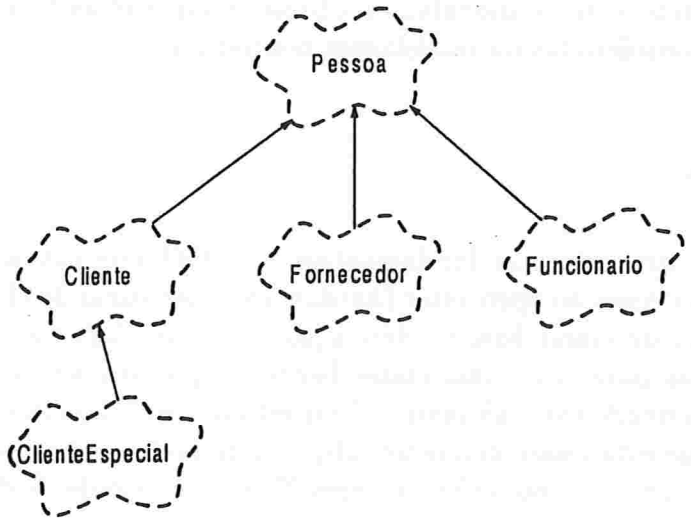

Figura 3.1: Modelo para cliente, fornecedor e funcionário como derivações de pessoa

- adicionar novos atributo ao objeto;
- adicionar novos atributos de classe;
- criar novas funções para substituir funções definidas na superclasse;
- criar novas funções (que não estão definidas na superclasse);

C++, por sua vez, permite que o comportamento definido para a classe base não seja disponível para usuários da classe derivada (através de derivações privativas).

Nierstrasz [Nier89] resume as diferenças entre as várias formas de herança através das seguintes questões:

A herança ocorre estática ou dinamicamente? Por exemplo, linguagens usadas em inteligência artificial constroem ou modificam a hierarquia de classes a medida em que o programa adquire conhecimento.

Quem são os clientes das propriedades herdadas? Deve-se considerar se as propriedades herdadas são de uso exclusivo para a implementação da classe derivada ou também funcionam como funções de interface para usuários da classe derivada.

Que propriedades podem ser herdadas? Devemos questionar quais membros podem ser herdados: atributos dos objetos, atributos da classe (membros estáticos), funções membro, etc.

Que propriedades herdadas são visíveis pelos usuários? Dos membros herdados, quais podem ser manipulados pelos usuários da classe derivada?

As propriedades herdadas podem ser modificadas ou suprimidas? A maioria das linguagens permitem a especialização de uma função da classe base na classe derivada. Em C++, por exemplo, definindo uma derivação como privativa, as propriedades da classe base tornam-se inacessíveis aos usuários da classe derivada.

Como os conflitos são resolvidos? Devido ao polimorfismo das linguagens orientadas a objetos, é possível a ocorrência de conflitos com relação à escolha da função que deverá ser executada com um dado comando. Numa hierarquia de classes, normalmente é considerada a função mais específica para o objeto que solicitou a execução, a função da classe ancestral mais próxima da classe do objeto. Com herança múltipla o problema torna-se bem mais complicado.

Um tipo especial de herança, denominado *herança múltipla*, ocorre com a declaração de uma classe derivada de duas outras classes ou mais. Este recurso, proibido em linguagens como Smalltalk, pode causar conflito de ambigüidade quando dois ancestrais ou mais apresentam a mesma função a ser executada pelo objeto derivado.

Classes Abstratas

Classes abstratas são classes criadas exclusivamente para serem derivadas e não caracterizam nenhum objeto do sistema, ou seja, durante toda a execução do programa, nunca existirão instâncias destas classes, que servem exclusivamente para descrever propriedades para serem herdadas.

Considere, por exemplo, a classe Iterador apresentada abaixo:

```
class Iterador {
public:
   virtual TipoBase *prox() = 0;
   virtual void inicio() = 0;
   virtual ~iterador() { };
};
```

Esta classe descreve operações para iterar sobre um conjunto, declarando funções para posicionar no início do conjunto e retornar o próximo elemento[2]. Como classes derivadas de Iterador, criaremos IteradorDeLista e IteradorDeVetor:

[2]A notação = 0 na declaração destas funções indica que elas, obrigatoriamente, devem ser implementadas nas classes derivadas.

```
class IteradorDeLista: public Iterador {
   Lista    *aLista;
   TipoBase *corrente;
public:
   IteradorDeLista(Lista *l): aLista(l)
      { corrente= NULL; }
   TipoBase* prox()
      { /* Retorna o proximo el. da lista */ }
   void inicio()
      { corrente= NULL; }
};

class IteradorDeVetor: public Iterador {
   Vetor *oVetor;
   int    corrente;
public:
   iteraVetor(Vetor * v ): oVetor(v)
      { inicio(); }
   TipoBase *prox()
      { return p[++corrente]; }
   void inicio()
      { corrente= -1; }
};
```

Esta derivação introduz um novo significado para herança: a classe base descreve um conceito abstrato que é implementado nas suas classes derivadas. Declaradas desta forma, instâncias das duas classes derivadas são manipuladas pelas mesmas funções (declaradas na classe base). Usando estas classes, podemos iterar sobre listas ligadas ou vetores sem nos preocupar se estamos manipulando uma ou outra.

3.3.5 Polimorfismo

Polimorfismo é o nome que damos à capacidade que um mesmo comando da linguagem tem para denotar operações distintas. Considere a classe Complexo apresentada abaixo (logicamente criada para manipular números complexos):

```
class Complexo {
    float real, imaginario;
public:
    Complexo(int r= 0, int i= 0);
    Complexo operator-(const Complexo c2) const;
    Complexo operator-();
    // ...
    void imprime();
};
```

Em um programa que utilize esta classe e também a classe Pessoa (declarada na página 35) pode apresentar o seguinte comando:

```
umObjeto.imprime();
```

Que função imprime() será executada? A função de números complexos ou de pessoas? Isto depende do tipo do objeto umObjeto:

- se ele for da classe Complexo será executada a função Complexo::imprime();
- se for da classe Pessoa, Pessoa::imprime().

O que ocorre é que o tipo do objeto que solicita a execução da função define o escopo da função que será executada. Polimorfismo denota esta característica de um mesmo comando poder assumir "formas" diferentes dependendo do contexto no qual aparece.

Chamamos de *sobrecarga de operadores* ao recurso que as linguagens oferecem para permitir a definição de um mesmo identificador para denotar funções ou operações distintas. Dizemos que a função imprime() foi *sobrecarregada* ou que houve *sobrecarga* desta função.

Também é possível sobrecarregar funções e operadores dentro do mesmo escopo desde que os parâmetros de cada uma das funções sobrecarregadas resolvam eventuais conflitos a respeito de que função deve ser executada com uma chamada. Por exemplo, na classe Complexo declaramos dois operadores $-$. O primeiro está definido como um operador binário e denota a operação de subtração de números complexos. O segundo é unário e denota a operação troca de sinal de um número. Toda operação de subtração de números complexos na qual somente um operando é especificado corresponde ao menos unário; as operações com dois operandos correspondem ao menos binário.

Notemos que a especificação de parâmetros de funções com valor default também definem uma forma de sobrecarga. No caso do construtor de Complexo, podemos utilizá-lo de três formas diferentes:

```
Complexo umComplexo;              //  umComplexo vale  0+0i
Complexo outroComplexo(3.14);     // outroComplexo vale 3.14+0i
Complexo umoutroComplexo(2,10); // umoutroComlplexo vale 2+10i
```

Acoplamento Dinâmico

Em C++ todos os conflitos referentes a que função deve ser executada em cada chamada pode ser decidida durante a compilação do programa. Entretanto, podemos mudar esse comportamento com a utilização de funções virtuais. *Funções virtuais* são funções declaradas na classe base de uma hierarquia de classes com a finalidade de fazer com que a escolha sobre qual função deva ser executada seja definida somente durante a execução do programa.

Por exemplo, a classe Objeto apresentada abaixo apresenta uma função, imprime(), declarada como virtual:

```
class Objeto {
public:
   // ...
   virtual void imprime() const {};
   // ...
};
```

Usando esta classe como base, vamos declarar duas classes derivadas, chamadas Filho1 e Filho2, que apresentam também uma função imprime():

```
class Filho1: public objeto {
public:
   void imprime() const
      { cout << "Sou Filho1\n"; }
};

class Filho2: public objeto {
public:
   void imprime() const
      { cout << "Sou Filho2\n"; }
};
```

Usando um ponteiro para a classe base, Objeto, podemos usá-lo para apontar para instâncias de suas classes derivadas:

```
main()
{
   Objeto *umObjeto;

   cout << "\n Responda: Sim ou Nao? (S/N)";
   if ( toupper(getchar()) == 'S' )
      umObjeto= new Filho1;
   else
      umObjeto= new Filho2;
   umObjeto->imprime();
}
```

Normalmente, poderíamos aguardar que o comando

```
umObjeto->imprime();
```

causaria a execução da função declarada na classe base, classe do ponteiro umObjeto. Como esta função, Objeto::imprime(), é virtual, a operação executada será aquela declarada na classe do objeto apontado. Desta forma, a escolha da função a ser executada neste programa só pode ser feita em tempo de execução.

Chamamos de *acoplamento dinâmico* ao mecanismo de escolha da função que será executada em tempo de execução de um programa.

3.4 Suporte à Reutilização

Uma das vantagens dos sistemas desenvolvidos com base em objetos que tem "seduzido" programadores e projetistas de sistemas é a facilidade com que as classes de uma aplicação podem ser reaproveitadas em outras. Chamamos de *reutilização de software* ao ato de aproveitar partes de um programa na composição de novas aplicações.

Em um resumo abordando as diversas formas de reutilizar software, Krueger [Krue92] apresenta 4 aspectos que devem ser considerados:

abstração Todas as abordagens para a reutilização de software usam alguma forma de abstração para os componentes de software. Abstração é a característica essencial em qualquer técnica de reutilização. Sem abstração, programadores seriam forçados a vasculhar uma porção de componentes reutilizáveis tentando descobrir o que cada um deles faz, quando estes componentes podem ser reutilizados e como reutilizá-los.

seleção A maioria das abordagens para implementar a reutilização auxiliam na localização, comparação e seleção de componentes. Por exemplo, métodos de classificação e catalogação podem ser usados para organizar uma biblioteca de componentes reutilizáveis e para auxiliar programadores quando estes buscam por componentes na biblioteca.

especialização Em muitas técnicas de reutilização, componentes similares são agregadas em um único componente genérico. Após selecionar um componente genérico para reuso, o programador especializa o componente através de transformações, definição de parâmetros imposição de restrições ou outras formas de refinamento. Por exemplo, uma implementação de pilha reutilizável deve aceitar como parâmetro seu tamanho máximo. Um programador que usa esta pilha genérica deve especializá-la fornecendo um valor para este parâmetro.

integração Técnicas de reutilização tipicamente têm um esquema de integração. Um programador usa este esquema para combinar uma série de componentes selecionadas e especializadas em um sistema.

Baseados nestes aspectos propostos, vamos avaliar como cada um destes quesitos são atendidos nos sistemas desenvolvidos usando objetos:

abstração De fato, esta é uma das características fundamentais de POO. Nos sistemas desenvolvidos com base em objetos, todo seu comportamento é implementado pelos objetos e estes implementam abstrações.

Deve-se, no entanto, tomar cuidado para não atribuir o mesmo nome a duas classes distintas (neste caso, se elas não implementam a mesma abstração, deve ter algo de "esquisito")[3].

Um detalhe simples que também influência o poder de reutilização de uma classe é o nome atribuído a ela. Por exemplo o nome Eledisse para identificar uma classe que manipula citações não é recomendado. Se a classe manipula citações, chame-a de Citacao [4].

seleção Esta atividade fica facilitada se cada classe criada for catalogada. A primeira regra básica é:

> Para selecionar um componente para reuso você deve saber o que ele faz.

[3]Esta "esquisitice" nem sempre é sinal de uma má modelagem. Por exemplo, um biólogo, ao consultar um catálogo com nomes de classes pode ser induzido a descrever uma abstração Bananeira como derivada de uma abstração que modele grafos conexos acíclicos.

[4]Infelizmente as linguagens de programação não aceitam um identificador como Citação.

Para efetivar a reutilização das classes em POO é importante evitar que um programador procure um componente examinando o código de cada classe criada. É fundamental catalogar as especificações de cada abstração descrevendo sucintamente o comportamento de cada classe. Daí estipulamos uma segunda regra básica:

> Para reutilizar um componente efetivamente você deve ser capaz de localizá-lo e recuperá-lo mais rápido que o tempo necessário para implementá-lo novamente.

especialização Na verdade cada classe descreve um molde para a criação de instâncias da classe. Desta forma o programador que reutiliza uma classe pode criar instâncias específicas para cada aplicação possível. Além de criar instâncias particulares através de parâmetros, ele pode também criar uma classe derivada de uma classe já pronta, especializando a abstração modelada pela classe base da derivação. Para aumentar o poder de reutilização das classes que criamos é essencial não criar uma classe para cada abstração identificada no problema, mas avaliar inicialmente a possibilidade de se definir uma hierarquia de classes para implementar a abstração.

Por exemplo, considere a classe Usuario que definimos neste capítulo. Para aquele modelo seria mais conveniente criar uma abstração para pessoa e definir usuário como uma especialização de pessoa. Desta forma, deixamos a disposição para reutilização uma componente - a classe Pessoa - que independe do domínio do problema e que pode ser reutilizada em outras aplicações.

integração Integração é imediata nos sistemas que amparam objetos, em C++ basta uma diretiva

```
#include<algo.h>
```

que o conteúdo do arquivo algo.h estará disponível nos nossos programas. Podemos até mesmo declarar classes com atributos e operadores homônimos que os compiladores de linguagens OO estão preparadas como resolver este conflito.

Outra restrição importante esta relacionada com as linguagens que amparam herança múltipla. Nestas linguagens a compilação pode ser abortada caso um classe derivada encontre a mesma função implementada em ancestrais de pais diferentes.

3.4.1 Utilização de Reutilização

Considere o problema de escrever um programa que leia uma seqüência de números, terminada por zero, e imprima a mesma seqüência na ordem inversa a de leitura.

Depois de analisá-lo chegamos às seguintes conclusões:

- é necessário armazenar os números digitados pelo usuário em um vetor (ou em outra estrutura de dados que se comporte como tal);

- este vetor deverá ser manipulado como uma pilha;

- o algoritmo para resolver o problema deverá ser algo da forma:

  - Inicializar uma pilha para armazenar os números lidos;
  - Ler e empilhar os dados digitados;
  - Desempilhar e imprimir os dados.

Lembrando que foi criada uma classe Pilha no inicio deste capítulo (na página 29), poderemos reutilizá-la para implementar este algoritmo. Supondo que a classe está declarada no arquivo pilha.h, o seguinte programa resolve o problema proposto:

```
#include <iostream.h>
#include "pilha.h"
void main()
{
    Pilha dadosLidos(50);
    int   num;

    do {
          cin >> num;
          dadosLidos.empilha(num);
    } while (num);
    while (!dadosLidos.vazia())
        cout << dadosLidos.desempilha();
}
```

Neste exemplo, basta que o programador localize o(s) arquivo(s) com a descrição da classe Pilha para compor o programa desejado.

Infelizmente a reutilização de classes não é "tão automática" quanto o exemplo anterior sugere. É costume que programadores em C++ descrevam cada classe em dois arquivos (um com a declaração da classe e o outro com a implementação). Por exemplo, considere a declaração da classe X descrita no arquivo X.h e listada abaixo:

```
#include "comum.h"
#include "Z.h"
#include "Y.h"
class X: public Y, public Z {
   int x;
protected:
   void atribuiX(int);
   int  retornaX();
public:
   X(int);
   ~X();
};
```

Um programador que queira utilizar esta classe deve localizar também as seguintes informações:

- os arquivos comum.h, Y.h e Z.h;
- o código das funções X::atribuiX(), X::retornaX(), X::X(int) e X:: X();
- os códigos das funções das classes Y e Z.

Além disto, o programador também deve considerar

- se os identificadores usados nos arquivos X.h, Y.h ou comum.h causam conflito com os que ele utiliza em seu programa;
- se algum destes arquivos referem-se a outros arquivos de inclusão;
- se o código destas classes já estiver no formato objeto, se eles são portáteis para a máquina que ele pretende usar;
- se estiver disponível o código fonte, se ele pode ser introduzido sem adaptações no ambiente em que programa.

3.4.2 Tipos Parametrizados

A declaração de *templates* em C++ fornecem um novo recurso para auxiliar na criação de tipos de dados genéricos. Da mesma forma que classes podem ser vistas como "formas" para a criação de instâncias, templates são formas para criação de classes. Por exemplo, a classe Pilha criada no início deste capítulo serve unicamente para manipular números inteiros; declarando uma template para implementar a mesma classe, teremos uma estrutura de dados muito mais útil:

```
#include<assert.h>

template <class TipoBase> class Pilha {
    TipoBase  *vetor;
    int        topo;
    int        tamanho;
public:
    Pilha(int tam= 100)
       { vetor= new TipoBase[tam]; topo= 0; tamanho= tam; }
    ~Pilha()
       { delete vetor; }
    char empilha(TipoBase elem)
       { if (topo==tamanho) return 0;
         vetor[topo++]= elem; return 1;
       }
    TipoBase desempilha()
       { assert(topo);
         return vetor[--topo];
       }
    char vazia()
       { return (!topo); }
    TipoBase elementoDoTopo()
       { assert(topo);
         return vetor[topo-1];
       }
};
```

A partir desta template Pilha podemos declarar pilhas de vários outros objetos e não somente de inteiros:

```
Pilha<int> umaPilhaDeInteiros;
Pilha<char> umaPilhaDeCaracteres;
Pilha< Pilha<int> > umaPilhaDePilhasDeInteiros;
```

Note que as operações utilizadas para implementar as funções da template Pilha devem estar definidas para todos os objetos usados como tipos básicos. No caso da pilha de pilhas de inteiros, por exemplo, deverá ocorrer um erro grave porque a função empilha(TipoBase) faz uma atribuição de objetos e o operador de atribuição não foi sobrecarregado para a classe Pilha.

3.5 Referências Bibliográficas

Para apreciar a evolução do modelo e dos diversos dialetos de POO, um bom começo é consultar textos referentes às linguagens que dão suporte a esta tecnologia. SIMULA tem como texto obrigatório [Birt73], escrito pelos criadores da linguagem. O projeto e a linguagem Smalltalk ganharam repercussão com a edição de agosto de 1981 da revista Byte [Byte81], quase que integralmente dedicada ao projeto. C++ surgiu da tentativa de se adicionar o conceito de classe à linguagem C [Stro83] e tem [Ellis90] como seu manual de referência.

As propriedades das linguagens orientadas a objetos foram pesquisadas em diversas fontes; [Taka90, Nier89, Booch91] apresentam uma visão geral sobre POO; a classe Iterador foi encontrada em [Stro93]; [Nyga86] fornece a visão de POO segundo os adeptos da linguagem Simula. Outros textos consultados foram [Pasc86, Thom89].

Um texto bastante completo sobre reutilização de software do qual usamos somente a parte relacionada com POO foi [Krue92]. Os problemas de localização de módulos para reutilização foram levantados em [Gibbs90].

Capítulo 4

Desenvolvimento Orientado a Objetos

Depois de distinguir os métodos de desenvolvimento baseado em ações dos métodos baseados em dados e de apresentar os recursos oferecidos pelas tecnologias baseadas em objetos, neste capítulo apresentaremos formas de se utilizar estes recursos no desenvolvimento de programas. Sem seguir um método de desenvolvimento específico, apresentaremos em linhas gerais problemas básicos que devem ser considerados em DOO, com especial ênfase na modelagem de dados. Para isto, após uma breve visão geral do problema de DOO, estudaremos como encaminhar o processo de desenvolvimento através de objetos. Para completar, no final serão apresentadas propostas de modelos de ciclo de vida para sistemas projetados com POO.

4.1 Introdução

Desde seu surgimento, POO tem divulgado uma série de conceitos e termos até então estranhos na área de computação. Depois do impacto inicial, notadamente marcado pela "edição smalltalk" da revista Byte [Byte81], começaram a surgir métodos adaptados a esta nova tecnologia propondo novas formas de abordagem para o problema de produção de software. Da formalização destes novas abordagens, surgem análise orientada a objetos e desenvolvimento orientado a objetos.

Embora já exista uma literatura bastante significativa a respeito do desenvolvimento de sistemas orientado a objetos, existe pouco consenso ou padronização entre a variedade de idéias propostas. Se por uma lado encontramos técnicas puramente orientadas a objetos, criadas com base nas características inovadoras e exclusivas deste paradigma, por outro, encontramos técnicas tradicionais com uma nova roupagem, fantasiando velhos métodos

para que estes se adaptem à nova tecnologia, e que também se autodenominam orientadas a objetos.

Embora este capítulo se proponha a abordar o desenvolvimento de sistemas orientado a objetos, não é possível ignorar completamente o processo de *Análise Orientada a Objetos* (AOO). Primeiro porque a fronteira separando estas duas fases não é nada definida, sendo que tarefas que alguns autores atribuem à análise, outros atribuem ao desenvolvimento. Segundo porque a própria classificação das técnicas propostas em técnica de análise ou desenvolvimento não é consensual.

4.2 Visão Geral

Devido à modularidade associada com a caracterização de objetos, os métodos de *Análise e Desenvolvimento Orientados a Objetos* (ADOO) em geral consistem na identificação e no gerenciamento das "partes" de um sistema, onde a cada parte corresponde uma classe. Cada uma destas partes pode ser tratada (e muitas vezes testada) individualmente de tal forma que o comportamento desejado do sistema surja naturalmente com a combinação de suas classes. Por causa desta decomposição granular de sistemas, muitos autores consideram como fundamentais as notações gráficas para descrever como se relacionam cada entidade componente do sistema (desta vez, o problema consiste no gerenciamento da complexidade da arquitetura do sistema resultante da análise e do desenvolvimento).

Monarchi [Mona92] apresenta um resumo de métodos de AOO, DOO e ADOO disponíveis e classifica estes métodos nas seguintes categorias: método de processo, representação ou ambos.

processo refere-se aos métodos procedimentais para realizar AOO, DOO ou algum aspecto particular relacionado a análise ou desenvolvimento, sem incluir diagramas ou notações para denotar o produto da análise ou desenvolvimento.

representação refere-se às notações gráficas ou diagramas utilizados para retratar o resultado da análise ou do desenvolvimento. O foco é centralizado na representação visual de algum resultado da análise ou desenvolvimento sem mostrar como obte-lo.

processo e representação inclui tanto os processos para dirigir a análise ou desenvolvimento como notações para retratar seus resultados.

Segundo esta classificação dos métodos obtém-se a tabela 4.1 na qual é retratado o escopo de uma série de métodos de ADOO.

CATEGORIA 1: Somente processo
Bulman
Henderson-Sellers e Constantine
Johnson e Foote
Scharenberg e Dunsmore
CATEGORIA 2: Somente representação
Ackroyd e Daum — Uma notação gráfica
Beck e Cunningham — Cartões de Classes/Responsabilidades/Colaboradores (CRC)
Cunningham e Beck — Diagramas de mensagens
Page-Jones e outros — Notação de Objetos Uniforme (NOU)
Wasserman e outros — Notação de Desenvolvimento Estruturado OO
Wilson — Diagramas de Classes
CATEGORIA 3: Processo e Representação
Alabiso — Transformação de Análise para Desenvolvimento (TAD)
Bailin — Método de Especificação de Necessidades Orientado a Objetos
Booch — Desenvolvimento Orientado a Objetos
Coad e Yourdon — Análise e Desenvolvimento Orientados a Objetos (AOO e DOO)
Gorman e Choobineh — Modelo Entidade-Relacionamento Orientado a Objetos
Livari — Uma estrutura para Identificação de Objetos
Kappel — Modelo Objeto/Comportamento
Lieberherr e outros — a Lei de Demeter
Meyer — Construção de Software Orientado a Objetos
Rumbaugh e outros — Técnica de Modelagem de Objetos
Shlaer e Mellor — Análise de Sistemas Orientada a Objetos
Wirfs-Broock e outros — Desenvolvendo Software Orientado a Objetos

Tabela 4.1: Classificação dos trabalhos de AOO e DOO

Independente do método considerado, para abordar todos os aspectos envolvidos na produção de um sistema projetado através de métodos orientados a objetos, a análise e o desenvolvimento devem considerar os seguintes pontos críticos:

Processo de análise do domínio do problema Uma técnica de análise de domínio geral que tem como objetivo a identificação do comportamento desejado do sistema e dos objetos (relacionados com o domínio do problema) que participam deste comportamento. Esta fase compreende as seguintes atividades:

1. Identificação de
   - classes relacionadas com o domínio do problema;
   - atributos que caracterizam cada uma destas classes;
   - comportamento esperado de cada classe;

- relacionamentos entre as classes identificadas (generalização, associação, agregação)

2. Acomodação de
   - classes
   - atributos
   - comportamento
3. Especificação do comportamento dinâmico do sistema (a inter-relação entre os objetos identificados)

Processo de desenvolvimento do domínio da solução Uma técnica de modelagem para o domínio da solução do problema (incluindo a especificação de objetos de interface com usuário e outros introduzidos para implementar os cenários propostos pelo processo de análise do domínio do problema). As atividades desta fase são:

1. Mesmo que em 1., 2. e 3. do processo acima para os seguintes tipos de classes:
   - classes relacionadas com a interface com o usuário
   - classes relacionada com a aplicação específica
   - classes base/utilitárias
2. Otimização de classes

Representação dos resultados obtidos na análise e no desenvolvimento Para representar os produtos obtidos na análise e desenvolvimento, as estruturas, funções e controle do sistema em diversos níveis de abstração. O ideal é que sejam obtidas descrições globais do sistema (macrorepresentações) além da representação das classes componentes (microrepresentação).

1. Visão estática de
   - objetos
   - atributos
   - comportamento
   - relacionamentos (generalização, agregação, associação, ...)
2. Visão dinâmica
   - comunicação entre objetos
   - controle/sincronização de objetos
3. Restrições
   - na estrutura (valores de atributo, cardinalidade de relacionamentos, etc.)
   - no comportamento dinâmico

Gerenciamento de complexidade Um mecanismo de abstração e gerenciamento de complexidade para particionar o problema e gerenciar a complexidade do sistema.

1. Mecanismos para gerenciar a complexidade estrutural conceitualmente
2. Mecanismos para gerenciar a complexidade comportamental conceitualmente
3. Representações do nível do sistema:
   - estrutura estática
   - comportamento/controle dinâmicos

Para nossa abordagem de DOO estudaremos em seguida os seguintes problemas associados com a produção de um sistema usando a tecnologia de objetos:

- Identificação de objetos
- Identificação de relacionamentos entre objetos
- Implementação de objetos
- Representação de objetos

Dentre os exemplos utilizados para ilustrar o processo de desenvolvimento apresentado no restante deste capítulo estaremos enfatizando aqueles relacionados com o estudo de caso que será resolvido no capítulo seguinte. O problema a ser resolvido é a implementação de um sistema de controle de empréstimos de uma biblioteca (detalhes sobre esta aplicação podem ser encontrados no referido capítulo).

4.3 Identificação de Objetos

Em qualquer aplicação orientada a objetos, as instâncias de classes compõem a maior parte do sistema e, se for utilizada uma abordagem OO "pura", todo o sistema consistirá de instâncias de classes. Por este motivo, o desenvolvimento de classes individualmente têm um impacto significativo na qualidade global da aplicação. Convencidos disto, uma atividade fundamental em POO é a caracterização de cada objeto componente do sistema.

Na análise do problema já é possível eleger possíveis candidatos a objetos. Objetos do domínio do problema representam elementos ou conceitos utilizados na descrição do problema; chamamos estes objetos de *objetos semânticos*.

objetos semânticos são objetos que possuem um significado no domínio do problema. Por exemplo, um sistema de registro de encomendas pode ter classes semânticas como Cliente, Item e Empregado.

O domínio da solução inclui os objetos semânticos, mas não somente eles. Durante o desenvolvimento, a ênfase está focalizada na definição de uma solução do problema. Novas classes semânticas podem ser incluídas no desenvolvimento na medida em que novas abstrações são descobertas. No exemplo de registro de encomenda, uma classe abstrata Pessoa pode ser criada como superclasse de Cliente e Empregado. Os outros objetos, estes pertencentes ao domínio da solução, podemos classificar nos seguintes tipos:

objetos de interface são associados com a interface com o usuário e não compõem diretamente uma parte do problema. Estes limitam-se apenas a fornecer a visão que o usuário terá dos objetos semânticos. Exemplos de tais objetos são classes que implementam sistemas de menus ou caixas de diálogos com usuário.

objetos de aplicação podem ser considerados como os "condutores" ou os "mecanismos de controle" do sistema. São eles que iniciam a aplicação e eventualmente controlam o seqüenciamento das funções de alto nível do sistema; nas linguagens procedimentais, correspondem ao programa principal. Nos sistemas OO, a classe de aplicação não deve fazer nada mais que controlar um menu ou iniciar a passagem de mensagens a outros objetos.

objetos base ou utilitários são componentes independentes da aplicação. A principal característica destes objetos é que eles são independentes tanto da aplicação quanto do domínio do problema. Por exemplo, objetos base podem ser agregados (vetores, cadeias de caracteres) ou números; um exemplo de objeto utilitário é um verificador de ortografia.

A identificação de classes corresponde a acomodar atributos e comportamento em classes que, posteriormente serão relacionadas através de relações como herança, agregação ou associação. A figura 4.1 descreve este processo: na medida em que identificamos atributos (na figura, denotados por A) e métodos (denotados por M), que podem ser semânticos, de interface, base/utilitários ou da aplicação (o tipo é denotado pelo índice n), eles são organizados em classes (que podem ser semânticas, de interface, base/utilitárias ou de aplicação) e estas serão posteriormente organizadas em estruturas descrevendo suas relações de herança, agregação ou associação.

4.3.1 Identificação de Objetos e AOO

Embora a fase de análise esteja fora do escopo deste texto, não é possível ignorá-la, pois é ela que inicia o processo identificação de abstrações que serão convertidas em classes. A seguir apresentaremos brevemente algumas técnicas de análise propostas para identificar objetos semânticos do sistema, a saber:

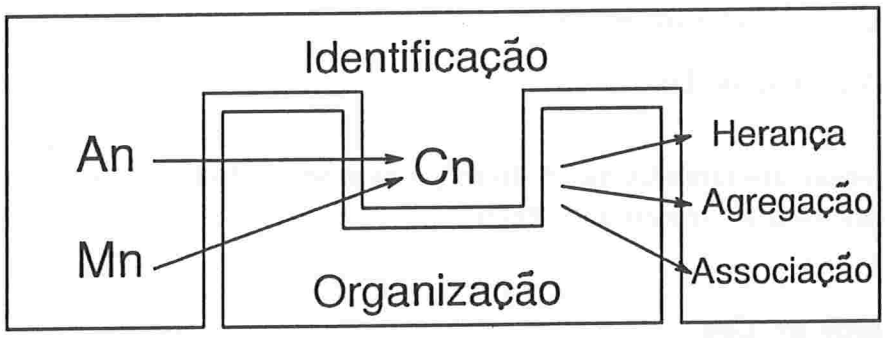

Figura 4.1: Processo de identificação de classes e seus relacionamentos

- Abordagens clássicas
- Análise de Casos de Uso
- Análise de Comportamento de Objetos
- Análise de Domínio

Abordagens Clássicas

As abordagens clássicas de análise OO consistem em "vasculhar" o domínio do problema procurando candidatos a objetos. Da mesma forma que desconfiamos imediatamente do mordomo em estórias de assassinato, na análise podemos (e devemos) desconfiar dos seguintes candidatos a objetos:

Coisas tangíveis carros, dados telemétricos, livros

Pessoas mãe, professor, cliente, mordomo

Eventos desembarque, interrupções

Interações empréstimos, reuniões

Por exemplo, o empréstimo de um livro deve ser modelado como um objeto? Por que não? Até mesmo as bibliotecárias anexam uma ficha referente ao livro no cadastro de um usuário para acusar um empréstimo. Modelando como um objeto, cada empréstimo apresenta as seguintes informações:

- Identificação do usuário que retirou o livro;

- Identificação do livro emprestado;
- Data da devolução do livro;

Desta forma, efetuar um empréstimo de livro passa a ser tratado como a criação de um objeto que modele esta abstração no sistema.

Análise de Casos de Uso

Uma abordagem para análise do sistema é identificar situações de sua utilização. Para isto, podemos descrever cenários que o sistema deve implementar. Cenários são descrições de situações que revelam possíveis usos do sistema.

Por exemplo, para um sistema de controle de empréstimos de uma biblioteca, os seguintes cenários são esperados:

- Um usuário solicita o empréstimo de um livro;
- Um livro novo é incluído no acervo;

A partir destes dois cenários já poderemos desconfiar imediatamente de candidatos a objetos como:

- usuário
- livro
- acervo

Estes objetos identificados podem ser apresentados em cartões CRC (apresentados na página 77).

Análise de Comportamento de Objetos

Análise de Comportamento de Objetos (ACB) é uma abordagem para a análise de sistemas orientada a objetos que apresenta uma descrição passo a passo de todos os aspectos da análise. O método propõe que a atividade de análise consiste na compreensão do comportamento do sistema para que este comportamento seja atribuído a partes do sistema. Nesta alocação de atividades, são identificados os responsáveis e colaboradores para cada comportamento apresentado pelo sistema. Os quatro passos do método são:

1. Compreenda a aplicação e identifique o comportamento do sistema.

   O resultado desta atividade é a caracterização dos comportamentos iniciais do sistema, as funções exercidas por ele.

2. Derive objetos usando a perspectiva comportamental.

   O resultado desta atividade é a identificação de objetos e comportamento destes objetos.

3. Comece a classificar objetos.

   Resulta na descrição de relações e coordenações de objetos.

4. Modele processos.

   Resulta especificação de necessidades e protótipo da análise.

O resultado desta análise é retratado em cartões CRC (veja página 77).

Análise de Domínio

Esta técnica de análise preocupa-se não somente com a modelagem de uma aplicação específica, mas com a criação de um modelo que possa ser reaproveitado em futuras aplicações dentro do mesmo domínio.

Por exemplo, no sistema de empréstimo de livros, deveremos modelar o conceito de livro do acervo. Na modelagem desta abstração, podemos criar uma representação específica para livros ou, usando herança, definir livro como herdeiro de outras classes para possibilitar futuras modelagens de conceitos como periódicos, relatórios técnicos, etc.

Aplicando este tipo de abordagem, será obtido um modelo conceitual para todo tipo de publicação e teremos, conseqüentemente o conceito de livro já modelado para outras aplicações dentro do mesmo domínio. Evidentemente, esta técnica exige a consultoria de um especialista que conheça todo o domínio do problema.

4.3.2 Diagramas de Estados de Objetos

Uma forma para caracterizar o comportamento de um objeto é considerar todos os estados que ele pode assumir durante sua "vida útil" dentro do sistema. Para capturar este comportamento, podemos usar um diagrama de estados de objetos.

Diagramas de estados de objetos é uma representação de uma máquina de estados finita na qual cada estado representa um possível estado que o objeto pode assumir e

as transições de estados correspondem a eventos do sistema que causam a mudança de estados do objeto. Estas máquinas são úteis para modelar e formalizar o comportamento de cada objeto no decorrer de seu ciclo de vida[1] dentro do sistema além de permitir a formalização de suas interconexões com outros objetos.

Considere, por exemplo, um cliente que se compromete a pagar uma mensalidade por um serviço mais os encargos decorrentes dos demais produtos consumidos. Suponha que para uma pessoa tornar-se cliente deste serviço tenha que fazer um depósito inicial igual a três vezes a taxa de mensalidade do serviço e a cada mês este cliente deve pagar sua fatura integralmente. Depois de um período inicial de um ano, se o cliente pagou suas contas assiduamente, ele terá a metade de seu depósito inicial reembolsada. Naturalmente, o cliente pode cancelar a qualquer instante o serviço e o restante da depósito inicial é reembolsado.

Para modelar o comportamento deste cliente, podemos construir um modelo de estados para representar seu ciclo de vida (veja figura 4.2). Ele poderá conter estados como *Tornando-se cliente*, *Em experiência*, *boa situação*, e *terminando o serviço.*

Usando este modelo, podemos formalizar o comportamento de um objeto e os eventos que infuem neste comportamento e usar esta informação para auxiliar na modelagem de uma classe. Quando um objeto for uma especialização de outro, seu modelo de estados apresentará o diagrama de estados de seu objeto base com a inclusão dos detalhes específicos que caracterizam os objetos da classe derivada. Assim, um diagrama de um objeto derivado poderá apresentar estados ou transições adicionais ou, até mesmo, estados e transições do diagrama do objeto base com comportamentos mais específicos.

4.3.3 Identificação de Objetos e DOO

Na fase de desenvolvimento, os objetos semânticos provenientes da análise são considerados e, possivelmente, refinados. O objetivo principal desta fase é estabelecer formas de implementar o funcionamento especificado na análise e para isto serão introduzidos novos objetos. Se na análise é observada a necessidade de um objeto para abstrair as propriedades de um acervo de biblioteca, com recursos para consulta ao seu conteúdo, empréstimos e inclusão/remoção de itens, cabe à fase de desenvolvimento descrever elementos que implementem este comportamento.

Nesta fase será necessário descrever a "forma" com que será implementado cada objeto semântico. Para isto, deverão surgir objetos que implementem a semântica de conjuntos (para modelar o conceito de acervo) e demais estruturas de dados necessárias.

[1]Aqui usamos o termo ciclo de vida com um significado diferente daquele usado para denotar as fases envolvidas na produção de um software. O ciclo de vida de um objeto caracteriza o comportamento que o objeto pode assumir durante sua vida útil dentro de um sistema.

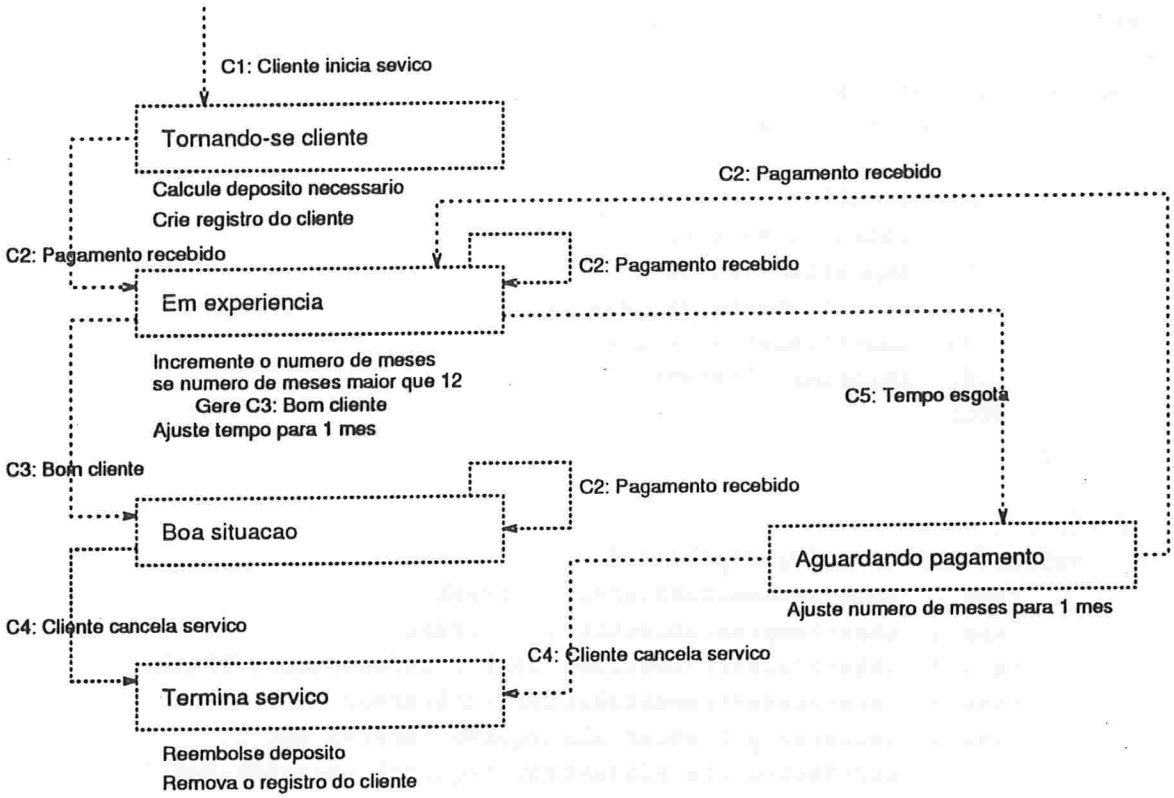

Figura 4.2: Modelo de ciclo de vida para clientes do serviço

Os outros objetos (de aplicação e base/utilitários) serão caracterizados também como resultados das especificações da análise:

- Os objetos de interface implementam a forma com que os usuários interagem com o sistema. Ao descrever o cenário "um usuário solicita a reserva de um livro", estaremos impondo um certo comportamento de entrada e saída de dados que poderá ser de responsabilidade de um objeto de interface. Na descrição e implementação deste objetos devem ser considerados os recursos gráficos oferecidos pelo computador e o tipo de comodidade que será oferecida aos usuários do sistema.

- Os objetos de aplicação simplesmente coordenam o serviço do sistema. A especificação destes objetos são baseadas nos serviços oferecidos pelo sistema (especificados pelos cenários) e sua função principal é "distribuir serviço" para os outros objetos.

  No código abaixo, é apresentado o construtor do objeto de aplicação para o sistema de empréstimos de biblioteca:

```
ServicoDeBiblioteca::ServicoDeBiblioteca()
{
  umAcervo.leAcervoDeArquivo("acervo2.dat");
  umCadastro.leCadastroDeArquivo("usuarios.dat");

  char *menuPrincipal[] =
       {  "(1): Consulta ao Acervo",
          "(2): Emprestimo De Livro",
          "(3): Cadastramento de Usuario",
          "(4): Cadastramento de Livro",
          "(5): Abandonar Sistema",
          NULL
       };

  for(;;) {
     switch( menu(menuPrincipal) ) {
         case 0: this->consultaAAcervo(); break;
         case 1: this->emprestimoDeLivro(); break;
         case 2: this->cadastramentoDeUsuarios(umCadastro); break;
         case 3: this->cadastramentoDeLivro(umAcervo); break;
         case 4: umAcervo.gravaAcervoEmArquivo("acervo.dat");
                 umCadastro.gravaCadastroEmArquivo("usuarios.dat");
                 exit(1);
         default:
            cout << "Opcao invalida\n";
     }
   }
}
```

Nesta classe, as únicas funções públicas são seu construtor e destrutor. Desta forma, para executar o programa que gerencia os empréstimos basta instanciar uma variável da classe ServicoDeBiblioteca:

```
ServicoDeBiblioteca umServicoDeBiblioteca;
```

4.4 Identificação de Relacionamentos

Em um sistema OO, objetos não atuam isoladamente. Direta ou indiretamente os objetos se relacionam para compor o funcionamento desejado do sistema. Na análise de um sistema, quando identificamos cenários como

- Um livro é incluído no acervo da biblioteca;

- Romeu gosta de Julieta;
- Vendedor consulta central de crédito para autorizar crediário a cliente;
- Uma lista ligada implementa um conjunto;
- Uma árvore é um vegetal.

estamos descrevendo relacionamentos entre (candidatos a) objetos.

Definir estes relacionamentos também faz parte do processo de identificação em ADOO e é a utilização deles que permite a criação de classes genéricas, adaptáveis e, conseqüentemente, reutilizáveis. Os principais tipos de relacionamentos são:

associação descreve uma relação descrevendo que vários objeto precisam se colaborar mutuamente para realizar alguma tarefa ou assumirem um certo comportamento (por exemplo, um cliente compra um automóvel).

generalização/especialização descreve uma relação de similaridade onde a subclasse implementa um caso particular da superclasse (por exemplo, um usuário de biblioteca é uma pessoa).

agregação descreve que algum atributo de um objeto é descrito por um outro objeto (por exemplo, motor é uma parte do automóvel).

Além destes relacionados acima que serão estudados com mais detalhes a seguir, encontramos outros tipos de relacionamentos:

instanciação Quando declarâmos uma instância de uma template também estamos relacionando classes.

utilização É um tipo particular de associação que descreve que para um objeto realizar uma atividade é necessária a colaboração de outro objeto.

metainformação Este tipo de relacionamento, existente em smalltalk, é usado para descrever uma classe cujas instâncias também são classes.

4.4.1 Herança

Herança deve ser usada para resgatar similaridades entre classes, sempre descrevendo uma relação do tipo generalização/especialização tal que a classe base da derivação possa ser vista como a descrição de um caso geral da classe derivada e a classe derivada, por sua vez,

possa ser vista como um caso específico da classe base. Toda instância da classe derivada é um elemento da classe base com algum comportamento mais específico (atributos ou operações adicionais ou específicos para objetos da classe derivada).

Um cuidado que devemos tomar é não mutilar o modelo conceitual na tentativa de relacionar quaisquer classes que apresentem atributos em comum. Por exemplo, em geral não devemos usar o fato de carros e frutas terem cor para modelar estes dois conceitos como herdeiros de uma classe Cor.

Por exemplo, considerando o exemplo do sistema de empréstimo de biblioteca, identificamos duas categorias de usuários:

usuários normais: são aqueles que se cadastram para utilizar a biblioteca, mas que não apresentam nenhum privilégio na utilização da mesma. Podem tomar emprestado livros do acervo sendo que cada livro deve ser devolvido uma semana após o empréstimo e não podem ter emprestado mais de dois livros de uma vez.

estudantes: são usuários que, por estudarem, têm o direito de retirar até quatro livros do acervo da biblioteca. No entanto, cada empréstimo tem validade também por uma semana.

professores são usuários que podem retirar até dez livros do acervo com um período de empréstimo de um mês para cada livro.

Aproveitando a similaridade entre estas abstrações devemos criar uma classe Usuario para ser usada como superclasse destas três categorias de usuários, UsuarioNormal, UsuarioEstudante e UsuarioProfessor. Desta forma, podemos acomodar todos os dados dos usuários como atributos da superclasse e deixar, para cada uma das subclasses a informação específica para cada uma destas categorias de usuários (apenas a informação referente aos privilégios com relação aos empréstimos que o usuário pode fazer).

Uma regra básica em POO é nunca se satisfazer completamente com o primeiro modelo de hierarquia de classes encontrado. A partir dele é possível compreender melhor o problema e encontrar falhas no modelo conceitual. POO é uma atividade de modelagem conceitual e o mecanismo de herança permite uma infinidade de interpretação e representação dos elementos do modelo. Uma boa hierarquia surge à medida que ganhamos experiência e senso crítico (tanto com relação ao modelo quanto com relação ao problema). As mudanças na hierarquia de classes definem um processo natural em POO que deve convergir a uma boa representação conceitual.

Por exemplo, repare que a hierarquia que obtivemos para modelar usuários da biblioteca pode ser refinado: a classe Usuário pode perfeitamente bem ser declarada como derivada da classe Pessoa que implementamos no capítulo anterior (pois todo usuário da

biblioteca é uma pessoa) e ficamos com a estrutura de classes para modelar o conceito de usuários de biblioteca como ilustrada na figura 5.1

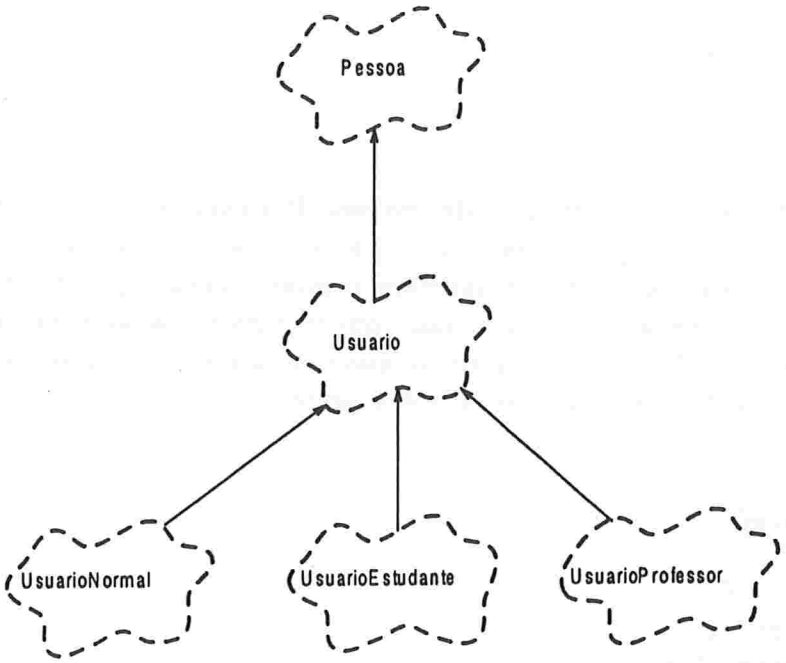

Figura 4.3: Hierarquia de classes para modelar o conceito de usuário de biblioteca

4.4.2 Agregação

Agregação descreve uma relação de posse, de todo/parte. Por exemplo, quando dizemos que uma pessoa tem um endereço, especificamos que a abstração pessoa tem um de seus atributos descrito pela abstração endereço. A importância desta relação na construção de sistemas OO reside na propriedade de encapsulamento apresentado pelas classes dos sistemas OO. Desta forma é possível modelar um objeto e seu atributo como se fossem entidades independentes e concentrar nossas atenções nas características essenciais do objeto e deixar detalhes internos do atributo para um tratamento posterior.

A seguir, apresentamos alguns casos nos quais é recomendável usar agregação:

Particionamento da descrição de um objeto Por exemplo, é muito mais simples dizer que uma pessoa tem nome, endereço e telefone que dizer que ela tem nome, rua, número da casa, cidade, estado, CEP, etc:

```
class Pessoa {
   char     *nome;
   char     *telefone;
   Endereco  local;
public:
   // ...
};
```

Escalonamento de desenvolvimento de classes Por exemplo, podemos, na fase inicial do desenvolvimento, considerar uma classe Endereco que não faz nada (como apresentada abaixo), servindo apenas para validar a declaração de um atributo de seu tipo na classe Pessoa e permitir que concentremos nossas atenções em "coisas mais importantes". No desenvolvimento, quando não for mais possível ignorar este detalhe, implementamos esta classe corretamente.

```
class Endereco{
public:
    Endereco() {}
    ~Endereco() {}
    void imprime() {}
};
```

Reutilização de classes Uma outra aplicação para agregação é separar da descrição de um objeto um conjunto de atributos que podem ser reaproveitados em outras aplicações. Por exemplo, na modelagem do conceito de automóvel, se descrevermos as características do atributo motor como um objeto isolado, este atributo pode ser reaproveitado posteriormente para descrever motores de eletrodomésticos.

4.4.3 Associação

Associação é uma relação semanticamente fraca porque não descreve uma característica do objeto. Esta relação descreve apenas as colaborações que os objetos necessitam, durante a execução do programa, para realizarem alguma atividade ou assumirem certo comportamento. Associação descreve a dependência entre objetos dentro do sistema.

Por exemplo, quando um usuário de biblioteca empresta um livro, nem ele e nem o livro estão mudando suas características. Para modelar o empréstimo de um livro, é criada uma instância da classe empréstimo (constando as identificações do usuário e do livro e a

data da devolução do livro); para associar o usuário com o livro que tomou emprestado, fazemos com que os dois objetos referenciem a mesma ficha de empréstimo:

```
class UsuarioDeBiblioteca: public Pessoa{
    ConjEmprestimos temEmprestado;
    // ...
public:
    // ...
    char anotaEmprestimo(Emprestimo*);
    void anotaDevolucao(Emprestimo*);
};

class LivroDeBiblioteca: public Livro{
   Emprestimo *emprestadoA;
public:
   // ...
   char empresta(Emprestimo*);
   void devolve();
};
```

Neste exemplo, o empréstimo funciona como um relacionamento entre dois conjuntos de entidades no modelo Entidade-Relacionamento (ER) de bancos de dados. Da mesma forma que no modelo ER, podemos classificar tipos de relacionamentos pela multiplicidade de objetos que relaciona. No exemplo acima, cada livro pode ser emprestado a um usuário enquanto que um usuário pode ter vários livros emprestados (obtemos assim um relacionamento "1 para N". Uma outra forma para se implementar este tipo de relacionamento é fazer com que um objeto contenha apontadores para os objetos com os quais está associado.

4.5 Estabelecimento da Interface das Classes

Depois de caracterizar uma classe devemos definir que operações devem ser oferecidas por suas instâncias aos usuários da classe.

Cada operação que um objeto realiza pode ser classificada em um dentre os seguintes tipos:

Inicialização Atribuição de um valor inicial ao objeto. Esta função costuma ser implementada por uma função especial, chamada de *construtor (da classe)* que é executada automaticamente com a criação de uma instância da classe.

Destruição Ajustes para desativar o objeto. Esta função também costuma ser implementada por uma função especial, chamada *destrutor*, que é executada automaticamente quando um objeto deixa de existir (quando ela sai de escopo, para as variáveis estáticas, ou quando é desalocada da memória, se for dinâmica).

Consulta Retornar atributos internos do objeto.

Modificação Modificar atributos internos do objeto.

Realização Realizar alguma operação específica.

A seguir são apresentadas algumas recomendações úteis para definir as funções de interface das classes:

- Em toda classe, a responsabilidade de atribuição de um valor inicial a uma instância é do construtor da classe. O primeiro passo para garantir que uma instância nunca assumirá um valor inválido durante a execução de um programa é certificar que ela já assume um valor válido logo após sua declaração. Para isto, use o construtor da classe.
- Evite o uso de classes amigas. Restringindo as funções que podem manipular os atributos da classe para as funções da própria classe faz com que os "culpados" pela atribuição de um valor inválido a uma instância da classe estejam limitados às funções da própria classe.
- Descrevendo as funções necessárias para a implementação do sistema, você resolve parte de um problema em um sistema; descrevendo um conjunto de funções que esgotam as possíveis manipulações do objeto, você cria uma classe com grandes chances de ser reutilizada. Claramente, as operações necessárias para um objeto dependem da aplicação e sua escolha deve ser muito bem planejada.
- As operações que uma classe descreve devem implementar alguma função ou comportamento que o conceito modelado assume no domínio do problema.

Um Exemplo: classe Data

Para exemplificar a descrição das funções de interface de uma classe, vamos descrever uma classe que represente datas. Nesta apresentação será necessário estabelecer um notação para apresentar as funções de interface da classe. Uma opção para esta notação é a linguagem com que o programa será implementado.

Nesta fase ainda não é necessário ter uma representação precisa de cada atributo; por este motivo, poderemos "relaxar" a sintaxe da linguagem de programação para que ela

acomode elementos ainda não determinados. Por exemplo, a única informação que temos sobre a classe data é que cada um destes objetos é descrito por três números:

- Um número representando o dia;
- um número representando o mês;
- e um número representando o ano.

Sem definir a representação destes atributos neste momento, "tentaremos" descrever a interface de data:

- São necessárias funções para inicializar cada instância desta classe e para isto, devemos usar o construtor da classe. Uma vez que podemos sobrecarregar esta função, criaremos um construtor para cada situação na qual ele é conveniente:

  Data(): Construtor default, utilizado para declarar uma instância de data sem que uma data fixa seja mencionada.

  Data(dia, mês, ano): Construtor com o qual especifica-se o dia, o mês e o ano.

  Note que o construtor default pode gerar um problema: que data deve ser atribuída aum objeto declarado desta forma? É importante que isto seja uma informação disponível aos usuários da classe. Podemos por exemplo, usar a data corrente (da execução do programa), usar uma data de referência (como 1/1/80) ou atribuir um valor inválido indicando que a variável não tem um valor atribuído (esta opção pode comprometer o desempenho das demais funções membro ao manipularem um objeto com valor inválido).

- Um operador importante é o que atribui o valor de uma variável da classe Data à outra:

  Data operador=(Data): Implementa a atribuição de valores entre datas.

  Em C++ toda classe tem um operador de atribuição declarado automaticamente. Se o implementador da classe definir um operador de atribuição, ele será usado. Se não definir, será usado um definido automaticamente que faz a atribuição copiando os atributos da classe membro a membro. No entanto, este operador definido automaticamente causa problemas se a classe contiver algum atributo que é alocado dinamicamente. Portanto, este operador deve ser definido (nos programas em C++) obrigatoriamente se a classe contiver um membro do tipo ponteiro.

  Um detalhe importante (em C++) é que se for necessário implementar o operador de atribuição então também é recomendável implementar um construtor de cópia (que apresenta o mesmo problema do operador de atribuição).

- Também é importante poder calcular se uma data ocorreu antes ou após a outra. Para isto poderíamos declarar um operador de ordem. Para criar uma classe mais útil, vamos declarar uma série completa de operadores de ordem e de comparação:

  booleano operador==(Data): Retorna verdadeiro se e somente se duas datas são iguais.

  booleano operador! =(Data): Retorna verdadeiro se e somente se duas datas são diferentes.

  booleano operador>(Data) Verifica se uma data é maior que outra.

  booleano operador<(Data) Verifica se uma data é menor que outra.

  booleano operador>=(Data) Verifica se uma data é maior que ou igual a outra.

  booleano operador<=(Data) Verifica se uma data é menor que ou igual a outra.

  Vamos também descrever funções para retornar as informações de datas (o dia, o mês e o ano):

  dia retornaDia(): Retorna o dia referente a data que chamou a função.

  mes retornaMes(): Retorna o mês da data.

  ano retornaAno(): Retorna o ano da data.

  Também é importante apresentar funções que alterem os atributos do objeto:

  atribuiDia(dia): Modifica o dia de uma data, atribuindo um novo valor.

  atribuiMes(mes): Modifica o mês de uma data, atribuindo um novo valor.

  atribuiAno(ano): Modifica o ano de uma data, atribuindo novo valor.

  Muitas vezes é necessário descrever uma data referente a um período após uma data de referência. Por exemplo, se um usuário de uma biblioteca empresta um livro pelo período de uma semana, a data de devolução deve ser a data do empréstimo mais sete dias. Para implementar este serviço definimos as seguintes funções:

  Data diaSeguinte(): Função que retorna a data referente ao dia seguinte a uma data.

  Data somaDias(int): Retorna a data obtida da data que chamou a função com um acréscimo de um certo número de dias.

  Data somaMeses(int): Retorna a data obtida da data que chamou a função com um acréscimo de um certo número de meses.

  Data somaAnos(int): Retorna a data obtida da data que chamou a função com um acréscimo de um certo número de anos.

  Note que especificamos nestas declarações um tipo booleano. Implementando estas funções em C++, como não temos um tipo booleano, teremos que usar um valor numérico que será tratado como booleano.

- operações para efetuar entrada e saída de datas

leData(): Faz a leitura de uma data de um dispositivo de entrada.

escreveData(): faz a escrita de uma data em um dispositivo de saída de dados.

Nestas funções não especificamos qual o dispositivo. Para fazê-las genéricas poderíamos passar o dispositivo como parâmetro. Implementando em C++, podemos sobrecarregar os operadores de inserção e remoção de streams.

Eventualmente pode estar faltando funções nesta lista (como, por exemplo, uma função que escreva uma data por extenso). A inclusão de uma nova operação em uma classe não altera em nada o comportamento das outras funções; uma operação adicional apenas define uma nova forma de manipulação do objeto.

Nas fases posteriores do desenvolvimento, deveremos caracterizar precisamente cada uma destas funções nos termos de uma linguagem de programação. Nesta hora deveremos que ter definidos a representação interna de cada objeto e os tipos dos parâmetros funções, bem como seu valor retornado.

4.6 Implementação das Classes

Após a identificação das classes e do estabelecimento de seus relacionamentos já teremos informações suficientes para iniciar a fase de implementação. Esta atividade é dependente da linguagem de programação que será utilizada e se resume na codificação do modelo de classe obtido nas fases anteriores.

É importante enfatizar que a passagem para esta fase não implica que o modelagem conceitual está concluída. A implementação de um protótipo pode revelar falhas na modelagem que, caso isto ocorra, deverá ser revista. Além disto, a facilidade que encontramos em POO de se descrever elementos genéricos e posteriormente refinar o comportamento destes elementos, incentiva a criação de protótipos antes do sistema final.

Nesta seção estudaremos alguns pontos críticos na implementação de cada classe do sistema.

4.6.1 Padrão Ortográfico

Ao escrever o código de um programa é importante manter um padrão na forma com que damos nomes aos identificadores utilizados na descrição de cada classe membros de cada classe.

Nome da classe Os identificadores de classes definem a visão que os usuários terão das classes; eles devem sugerir a abstração que a classe implementa. Quando encontramos uma classe chamada Pilha já somos induzidos a imaginar que ela implementa uma pilha [2].

Nomes dos demais identificadores Os nomes dos atributos de cada classe não são da conta dos usuários da classe! Porém é importante que estes identificadores tenham nomes significativos (considerando-se a característica que eles abstraem) pois reformulações no modelo conceitual pode exigir modificações internas nas classes e nomes decentes facilitam este trabalho.

Já os nomes das funções de interface devem exprimir que tipo de operação elas implementam. Para atribuir estes nomes, não devemos ter preconceitos contra nomes longos; o mais importante é que eles descrevam a semântica da operação. Muitas vezes a escolha de um identificador envolve reflexões nada simples.

Por exemplo, dado um objeto para manipular datas, digamos Data, que nome daríamos a uma operação que soma uma data com um certo número de dias? No caso do sistema de empréstimos de biblioteca, será necessário que se anote que um livro será devolvido em uma semana. Usando o símbolo de soma, poderemos ter nos programas comandos como

```
Data devolucao, hoje;
// ...
umaData= hoje+7;
```

para denotar a soma de datas com dias. Resolvido este problema, como denotaríamos a operação de soma de uma data com o número de meses (pois podem existir usuários que emprestem livros pelo período de um mês). Novamente precisaremos de um operador que some uma data com um número inteiro.

Por este motivo, no caso de datas foi considerada a criação de funções, chamadas de somaDias, somaMeses e somaAnos, sem utilizar o operador de soma. Também é altamente recomendável manter um padrão nos nomes de identificadores dos programas.

4.6.2 Definição da Representação dos Atributos da Classe

Este tipo de discussão já é bastante conhecida por pessoas com alguma formação básica em estruturas de dados e não temos muitas particularidades a acrescentar.

Um recurso que encontramos em POO é a descrição de atributos de uma classe como uma instância de outra (que chamamos de agregação). Com agregação é possível apresentar um atributo de uma classe sem descrever sua representação interna.

[2] Embora este nome não indique se a classe modela pilhas de rádio ou uma estrutura de dados que os computólogos usam para exemplificar o conceito de tipos de dados.

Outro recurso que temos a disposição é a declaração de atributos que são compartilhados por todas as instâncias da classe (chamados de *membros estáticos*). Por exemplo, para modelar as categorias de usuários de biblioteca, podemos declarar 3 classes derivadas de Usuario, uma para cada categoria. Note que a única distinção entre estas categorias é o número de livros que o usuário pode tomar emprestado e o tempo de duração de cada empréstimo. Como estes atributos tem o mesmo valor para todos as instâncias da classe, não é conveniente declará-los como os outros atributos. Podemos defini-los como membros estáticos, e teremos uma mesma variável que se comporta como se fosse membro de todas as instâncias da classe.

Membros públicos ou privativos? Uma lei que devemos obedecer sempre é nunca declarar qualquer atributo de uma classe como público. Se um usuário da classe necessita consultar ou alterar o valor de um atributo, ele deve procurar uma função específica para este fim. Se não houver tal função deve-se considerar as seguintes justificativas:

- a manipulação desejada é indevida para o objeto (considerando a modelagem conceitual);
- a classe não foi projetada com o objetivo de ser reutilizada;
- o projeto da classe "ainda não é perfeito".

4.6.3 Definição das Funções de Interface da Classe

Para esta atividade, primeiro devemos especificar precisamente o tipo do valor retornado pela função e também dos parâmetros usados na sua chamada. Uma vez que podemos definir várias funções usando o mesmo identificador, devemos também avaliar quais opções de parâmetros é conveniente deixar a disposição dos usuários da classe. Daí em diante a implementação torna-se imediata.

Um detalhe importante é que as funções de interface da classe devem ser seus únicos membros públicos. Note que uma classe pode conter funções que não sejam de interface pois para a implementação destas funções de interface, devemos ser sistemático e metódicos [3]. Quando for necessário criar funções auxiliares para implementar uma das funções membro, estas funções devem ser declaradas como privativas da classe.

Exemplo: Classe Data

A seguir apresentamos a parte de declaração da classe Data modelada anteriormente:

[3] Nesta hora, programação estruturada é uma boa alternativa

```
class Data{
    int  dia, mes, ano;

    char eUltimoDiaDoMes();
    char eUltimoDiaDoAno();
public:
    Data();
    Data(int, int, int);
    char retornaDia() const;
    char retornaMes() const;
    char retornaAno() const;
    Data somaDias(int) const;
    Data somaMeses(int) const;
    Data somaAnos(int) const;
    Data diaSeguinte();

    int operator==(const Data& outra) const;
    int operator!=(const Data& outra) const;
    int operator>(const Data& outra) const;
    int operator<(const Data& outra) const;
    int operator>=(const Data& outra) const;
    int operator<=(const Data& outra) const;

    friend istream& operator>>(istream&, Data&);
    friend ostream& operator<<(ostream&, const Data);
};
```

Não será apresentada a implementação destas funções por serem simples e óbvias demais. Note ainda que foram incluídas duas funções não previstas (eUltimoDiaDoMes() e eUltimoDiaDoAno()). Estas funções foram criadas para facilitar a implementação da função diaSeguinte() e foi considerado que estas funções adicionais não deviam constar na interface da classe. Por este motivo elas estão declaradas como funções privativas da classe.

4.7 Qualidade do Modelo

A qualidade de um modelo baseado em objetos depende do domínio do problema, tornando-se difícil apresentar regras gerais para avaliar o modelo. Por exemplo, a seguir são apresentadas algumas recomendações úteis, tomadas de [Kors90] para nortear o desenvolvimento das classes de um sistema OO:

1. Os únicos membros na parte pública da classe devem ser os operadores da classe.

2. Uma instância de uma classe A não deve enviar uma mensagem diretamente um componente da classe B.
3. Um operador deve ser público se e somente se for disponível aos usuários da classe.
4. Cada operador pertencente a uma classe deve deve modificar ou acessar algum dado da classe.
5. Uma classe deve ser dependente de um número mínimo de classes possível.
6. A interação entre duas classes deve ser explícita.
7. Cada subclasse deve ser desenvolvida como uma especialização da superclasse, com a interface pública da superclasse tornando-se uma parte da interface pública da subclasse.
8. A classe raiz de uma estrutura de herança deve ser um modelo abstrato do conceito modelado.

No entanto, o número de dependências de uma dada classe é conseqüência do domínio do problema. Uma recomendação também comum é que uma subclasse deve especificar um subtipo da classe da qual foi derivada.

4.8 Representação

Através de representação estamos considerando qualquer recurso gráfico utilizado para exprimir algum resultado do desenvolvimento do sistema[4].

Com POO, tornam-se fundamental tais recursos, pois mesmo sistemas simples, desde que bem modelados, são representados por várias classes relacionadas entre si e que coordenam o funcionamento do sistema.

[Fowler93] considera três visões básicas de um sistema: as visões de dados, de comportamento e arquitetural.

Visão de Dados A visão de dados (ou estrutural) de um sistema concentra-se na descrição dos tipos de objetos (classes) que compõem o sistema e dos vários tipos de relacionamentos estáticos que existem entre eles. Existem três tipos principais de relacionamentos entre classes: associação, generalização e agregação.

A visão de dados é apresentada numa forma parecida com o modelo Entidade-Relacionamento, embora as notações correntes variem bastante.

[4]Embora não tenhamos citado anteriormente, os diagramas que estamos utilizando normalmente para representar classes são tomados do método de Booch.

Visão de Comportamento Segundo o próprio Fowler, a visão de comportamento descreve como um sistema muda. Idealmente, esta descrição deveria ser executável, ou seja, pelo menos na teoria, um compilador poderia ser escrito para ela e a versão compilada seria executável. Esta visão é focalizada nas seqüências de ações que mudam o retrato estático definido na visão de dados. A abordagem mais comum para a modelagem de comportamento ainda é o *fluxograma.*

Visão Arquitetural Combinadas, as visões de dados e de comportamento podem proporcionar uma descrição completa de um sistema. Infelizmente, ela torna-se facilmente complexa demais para ser compreensível. Uma técnica arquitetural pode dividir um sistema em subsistemas. Métodos estruturados tradicionais valem-se desta técnica através da decomposição funcional.

4.8.1 Representação de Classes e Relacionamentos

Historicamente, a forma mais simples e tradicional de se descrever um sistema modelado através de objetos é a descrição da hierarquia de derivação das classes que compõem o sistema. O próprio ambiente do sistema Smalltalk já apresentava um utilitário para consulta desta hierarquia de classes. Este utilitário descrevia todo o sistema através de uma árvore com as seguintes propriedades (veja a figura 4.4):

- Os nós representam as classes incluídas no sistema;
- A raiz da árvore é a classe *objeto*;
- Um nó A é filho de um nó B, se a classe referente ao nó A for descendente direta da classe referente ao nó B.

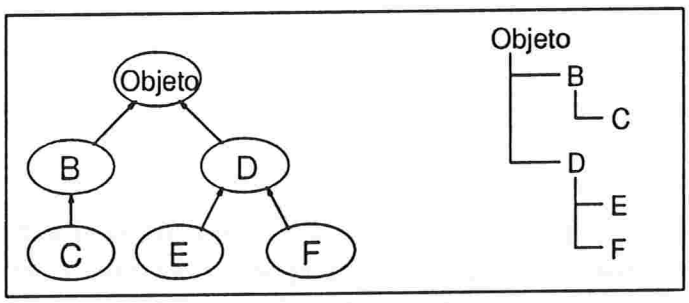

Figura 4.4: Arvores representando hierarquias entre classes

Infelizmente esta representação é insuficiente para a caracterização completa de um sistema. Ela limita-se a descrever os dados sem apresentar como os objetos interagem.

Além disto, apresenta apenas o resultado final do desenvolvimento: depois de completar (uma versão de) um sistema, obter estes diagramas é uma tarefa trivial.

4.8.2 Cartões CRC

Cartões CRC (Classes/Responsabilidades/Colaborações) emergiram como um meio de simples e maravilhosamente efetivo para analisar cenários. Proposto inicialmente por Beck e Cunningham como uma ferramenta para ensino de POO, estes cartões provaram ser uma ferramenta útil de desenvolvimento que facilita o debate de idéias e aumenta a comunicação dentro da equipe de desenvolvimento.

Um cartão CRC consiste de um cartão indexado de papel (por exemplo, com 10 cm por 7 cm), no qual o analista escreve - a lápis - o nome de uma classe (na parte superior do cartão), suas responsabilidades (na parte esquerda) e seus colaboradores (na parte direita), como ilustrado na figura 4.5.

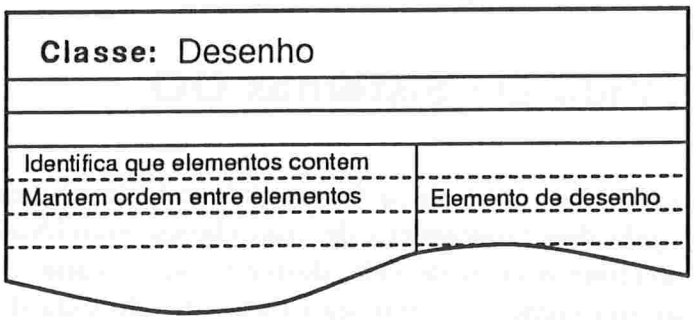

Figura 4.5: Detalhe de um cartão CRC indexado com colaboradores

Um destes cartões é criado para cada classe considerada relevante ao cenário. A medida que a equipe de análise investiga os cenários resultantes da análise, ela pode atribuir novas responsabilidades a uma classe existente, agrupar certas responsabilidades para formar uma nova classe, dividir as responsabilidades de uma classe entre classes mais refinadas ou atribuir estas responsabilidades a uma outra classe.

4.8.3 Outros Diagramas

Em POO são necessárias, principalmente, formas para representar classes e seus relacionamentos (uma vez que o sistema é particionado em classes). Dependendo do método de DOO empregado, um diagrama específico torna-se essencial para retratar o resultado do método. Aqui não aprofundaremos a exposição destes destes diagramas.

Alguns autores sugerem diagramas adicionais, como, por exemplo:

- Diagramas para retratare o comportamento dos objetos do sistema (que mensagens envia e para quem as envia).
- A representação de ciclo de vida de um objeto apresentado na página 59 também fornece uma notação para descrever uma entidade do sistema.
- Mesmo usando objetos, formas tradicionais de representação (como diagramas do modelo entidade-relacionamento ou diagramas de fluxo de dados) também são recomendadas por alguns autores.
- Diagramas de módulos fornecem subsistemas envolvidos na arquitetura de um sistema. Por exemplo, podemos descrever módulos de entrada e saída de dados, interface com usuário, etc. O ideal destes diagramas é descrever que partes estão envolvidas no funcionamento do sistema independentemente das classes contidas em cada subsistema.

4.9 Ciclo de Vida De Sistemas OO

Conforme vimos no decorrer deste capítulo, o desenvolvimento de sistemas usando objetos pode ser caracterizado pelo desenvolvimento de suas classes individualmente. Por este motivo, não é correto analisar o ciclo de vida destes sistemas como o ciclo de vida de um projeto isolado, mas sim como a combinação dos ciclos de vida de uma coleção de pequenos projetos isolados.

Um outro detalhe importante é que POO permite a criação rápida de protótipos de tal forma que viabiliza a implementação de um modelo simplificado inicial capaz de ser expandido até atender a todas as especificações do sistema. Desta forma, POO permite um intercâmbio entre as fases de análise, desenvolvimento e implementação, de tal forma que uma fase valida ou revela deficiências da outra.

DOO é uma atividade em que classes são desenvolvidas e testadas independentemente, paralelamente. Enquanto que classes como Data (que vimos na página 73 são facilmente identificadas e caracterizadas nas fases iniciais do projeto, uma classe que modela o conceito de usuário de biblioteca exige uma análise mais completa que termina por gerar não só uma única classe, mas um conjunto de classes relacionadas por herança. Possivelmente, a classe Data já poderá estar implementada enquanto a equipe de desenvolvimento ainda está discutindo "o melhor" modelo conceitual para usuários.

Para um paradigma que está alicerçado na modularidade de seus produtos e no grande poder de reutilização destes módulos, parece estranho analisar seu ciclo de vida através

de modelos "uniformes", que tratam o sistema desenvolvido como um todo. Mesmo que a vida útil de um sistema OO seja limitada, não podemos garantir o mesmo a respeito dos módulos que o compõem. A implementação de classes genéricas permite a criação de módulos independentes do domínio da aplicação que podem ser incluídos em bibliotecas de componentes reutilizáveis.

Por exemplo, no seu método de desenvolvimento, Booch combina dois modelos de ciclo de vida para a produção do sistema:

macroprocesso: no macroprocesso é tratado o desenvolvimento do sistema como um todo, envolvendo as seguintes fases:

conceitualização: estabelecimento das necessidades essenciais do sistema;

análise: desenvolvimento de um modelo do comportamento desejado para o sistema;

desenvolvimento criação de uma arquitetura para o modelo resultante da análise;

evolução expansão da implementação;

manutenção Gerenciamento da evolução após entrega.

microprocesso: esta atividade compreende a manipulação dos objetos do sistema, considerando o tratamento de cada classe e seus relacionamentos. São as atividades desta fase:

- Identificar as classes e os objetos em um dado nível de abstração;
- Identificar a semântica destas classes e objetos;
- Identificar os relacionamentos entre estas classes e objetos;
- Especificar a interface e, então, a implementação destas classes e objetos.

Enquanto o estágio final de um sistema é sua utização, devemos considerar que o estágio final de uma classe é sua inclusão em uma biblioteca de componentes.

Uma proposta de modelo de ciclo de vida para sistemas desenvolvidos desta forma é *engenharia concorrente* (EC). O Instituto para Análise de Defesa define EC da seguinte forma: "Engenharia Concorrente é uma abordagem sistemática para o desenvolvimento integrado e concorrente de produtos e seus processos relacionados, incluindo manufatura e suporte". Em outras palavras, EC é uma abordagem para se desenvolver simultaneamente diversas facetas de um mesmo sistema. EC pode ser considerada como uma alternativa à utilização de um modelo de cascata de grande porte no qual toda análise é completada antes de começar o desenvolvimento e todo desenvolvimento é completado antes de se iniciar a implementação. No lugar deste modelo, podemos considerar diversas mini-cascatas

operando em paralelo, uma para cada parte dos sistema, que pode compreender de uma classe a um conjunto de classes relacionadas.

Aqueles que preferem o modelo em espiral para o desenvolvimento de sistemas, EC tem implicações similares. No lugar de uma espiral de desenvolvimento linear e progressivo, subgrupos de espirais concorrentes podem ser propagados com projetos de EC.

4.10 Referências Bibliográficas

Um texto usado como referência básica para este capítulo foi [Mona92], que forneceu a visão geral dos métodos de DOO. Análise orientada a objetos foi consultada em [Booch91], enquanto que o ciclo de vida de um objeto foi pesquisado em [Shlaer92], do qual também foi retirado o exemplo do diagrama para o cliente.

Modelos de ciclo de vida para sistemas desenvolvidos com objetos foram consultados em [Booch91, Odell92, Hender90].

Capítulo 5

Estudo de Caso

Neste capítulo estudaremos a aplicação do método de DOO para a solução de um problema concreto. Abrangendo da modelagem conceitual à implementação, nosso objetivo é aplicar as idéias apresentadas nos capítulos anteriores para verificar sua validade. O problema proposto, que teve alguns de seus aspectos usados como exemplos de capítulos anteriores, é o de implementar um sistema para controlar empréstimos dos livros de uma biblioteca.

5.1 O Problema

Inicialmente devemos definir precisamente o problema que devemos resolver:

> Devemos implementar um sistema que controle empréstimos de livros de uma biblioteca. Este sistema deve oferecer os seguintes serviços:
>
> - Gerenciar o cadastro de usuários da biblioteca, implementando opções para efetuar a inclusão e remoção das informações de usuários do cadastro, bem como a alteração de dados cadastrais;
> - Gerenciar o cadastro de livros (que armazena a descrição dos livros presentes no acervo) fornecendo opções de inclusão e remoção de livros do cadastro;
> - Oferecer recursos para os usuários consultarem a presença de livros no acervo (buscando pelo título do livro ou pelo autor da obra);
> - Controlar empréstimos dos livros do acervo a leitores cadastrados.
>
> Teremos ainda que tratar diferentemente três categorias distintas de usuários:

usuários normais: são usuários que não gozam de grandes privilégios de tratamento na biblioteca: podem retirar dois livros no máximo de cada vez pelo período de uma semana de empréstimo;

estudantes: são usuários que podem tomar até quatro livros emprestados pelo período de 2 semanas;

professores: são usuários que podem retirar até dez livros emprestados pelo período de um mês por cada livro.

5.2 Análise Inicial

5.2.1 Restrições do Domínio

Uma aplicação como esta deve manter bases de dados com informações cadastrais sobre usuários da biblioteca e livros catalogados. No entanto, implementar um sistema gerenciador de base de dados bem feito exige um esforço que está fora das propostas deste trabalho (mas que para uma aplicação comercial deve ser considerado). Conformados com isto, simplificaremos as exigências com relação ao armazenamento de dados cadastrais armazenado-os na memória do computador durante a execução do sistema. Da mesma forma, estaremos supondo que este sistema é executado em um computador individualmente (enquanto que para uma aplicação comercial seria interessante um aplicativo para rede de computadores).

Não faremos, também, tratamento específico para descrever diversos tipos de publicações. Estaremos preocupados apenas com livros, ignorando artigos, revistas, etc. Uma outra restrição adotada para simplificar o problema, será considerar apenas um autor para cada item do acervo.

5.2.2 Funções do Sistema

A partir da descrição do problema, concluímos que este sistema deve oferecer quatro serviços básicos:

consulta: responsável pela exibição dos itens presentes no acervo para serem emprestados;

empréstimo: atendimento às solicitações de empréstimo de livros a usuários;

cadastro de usuário: responsável pela coleta de informações pessoais do usuário e sua inclusão no cadastro de usuários da biblioteca;

cadastro de livros: responsável pelo gerenciamento (inclusão e remoção) de itens no acervo.

5.2.3 Cenários Previstos Para o Sistema

Usando o método de análise baseado em casos de uso, apresentado no capítulo anterior, encontramos os seguintes cenários relacionados com a utilização do sistema que devemos produzir:

- Um usuário solicita o empréstimo de um livro do acervo;
- Um usuário devolve um livro que tomou emprestado;
- Um usuário consulta a existência de um livro no acervo;
- Uma pessoa cadastra-se como usuária;
- Um livro é incluído no acervo;
- Um livro é excluído do acervo.

Além destes, obtidos imediatamente com a caracterização do problema, podemos identificar os seguintes, adicionais:

- Um usuário solicita o empréstimo de um livro não disponível;
- Um usuário não devolve um livro que tomou emprestado;
- Um usuário modifica seus dados cadastrais;
- um usuário consulta a presença de livros de um dado autor no acervo da biblioteca.

5.2.4 Identificação das Primeiras Abstrações

Fazendo a análise dos cenários que encontramos acima, costatamos de imediato que o sistema deve modelar abstrações de usuários da biblioteca, livros, acervo da biblioteca e cadastro de usuários da biblioteca:

Usuário: apresenta as características relevantes de um usuário da biblioteca:

- apresenta uma identificação (dados pessoais);
- trata-se uma pessoa cadastrada como usuário da biblioteca;
- cada usuário é classificado em uma das categorias de usuários da biblioteca (que pode ser professor, aluno ou funcionário);
- tem o direito de solicitar empréstimos de livros;
- seus dados cadastrais devem incluir informações a respeito de empréstimos pendentes.

Livro: modela o conceito de livro, apresentando as seguintes propriedades:

- apresenta uma identificação (dados bibliográficos);
- está presente no acervo da biblioteca;
- deve ser possível verificar se o livro está emprestado ou não;
- pode ser emprestado a usuários.

Acervo: trata-se de uma abstração que representa o conjunto de livros do catálogo da biblioteca e tem como propriedades:

- apresenta operações de conjuntos (inclusão e remoção de livros);
- deve ser possível consultar a existência de um dado livro no acervo.

Cadastro De Usuários: modela o conceito de cadastro dos usuários com permissão para retirarem emprestados os livros do acervo da biblioteca. Este conceito tem como propriedades:

- apresenta operações de conjuntos (inclusão e remoção);
- deve ser possível consultar a existência de um dado usuário no cadastro.

A partir da descrição destas abstrações iniciais, reconhecemos a necessidade de uma estrutura de dados para modelar o conceito de conjunto (pois o acervo é um conjunto de livros e o cadastro de usuários é um conjunto de fichas referentes a usuários). Além desta nova abstração, devemos decidir "como" acusar um empréstimo de um livro a um usuário. Uma solução é descrever este serviço também como uma abstração:

Conjunto: modela o conceito de conjunto, apresentando recursos para armazenar elementos e operações básicas para sua manipulação:

- deve ter capacidade para armazenar elementos (de preferência de qualquer tipo);

- deve apresentar operações para sua inicialização como conjunto vazio e inserção e remoção de elementos;
- apresentar recursos para permitir a iteração sobre seus elementos.

Empréstimo: modela o conceito do serviço de empréstimo de um livro a um usuário da biblioteca.

- identificação do usuário que emprestou o livro e do livro emprestado;
- data de devolução do livro.

5.3 Identificação de Classes e Relacionamentos

O conhecimento das abstrações básicas já obtidas (usuário, livro, acervo, cadastro de usuários, empréstimo e conjunto) permite que passemos a desconfiar de uma série de candidatos a objetos.

Cada uma daquelas abstrações forneceria imediatamente um objeto se não fôssemos metódicos e sistemáticos (segundo o modelo de programação que adotamos). Devemos analisar cuidadosamente cada abstração obtida e descrever não necessariamente um objeto para cada abstração, mas sim uma estrutura de objetos que modele (bem) o conceito abordado. Para esta fase é recomendável apresentar cada classe encontrada em um cartão CRC para particularizar suas funções e colaboradores dentro do sistema.

Desta forma, vamos analisar cada uma daquelas abstrações com mais detalhe, tentando caracterizar estes objetos e já procurando relacionamentos básicos.

Livro Os livros do acervo da biblioteca são os itens que os usuários eventualmente retiram por empréstimo. Todos os livros da biblioteca são catalogados e podem ser emprestados desde que estejam disponíveis (claramente, não é possível retirar um livro que está emprestado a outra pessoa). Cada livro apresenta informações para sua identificação (código, título, autor, editora e ano da edição) além de informações descrevendo sua situação (se está emprestado ou não).

A modelagem de livro torna-se simples se não nos preocuparmos com a natureza dos diversos tipos de publicação existentes (como revistas, relatórios técnicos, artigos, etc.). Como restringimos nosso modelo para livros especificamente, basta uma classe que modele o conceito de livro e outra, derivada de livro, que descreva particularidades dos livros do acervo da biblioteca (como um código de identificação, a data de aquisição, etc.).

Assim, identificamos as duas primeiras classes para o nosso sistema:

- Livro, classe que modela o conceito de livros genéricos;
- LivroDeBiblioteca, classe derivada de livro que modela o conceito específico de livros de uma biblioteca.

Acervo O acervo de uma biblioteca é o conjunto formado pelos seus livros. Desta forma, esta abstração deve descrever um conjunto (com operações como inclusão, remoção e consulta de elementos) de abstrações de livros de biblioteca. Devemos descrever o conceito de conjunto e apresentar acervo como um caso particular. Pensando no reaproveitamento deste modelo em outras aplicações, o ideal é dirigir o desenvolvimento da representação de conjuntos de forma a produzir uma classe que sirva para tratar de conjuntos genéricos, que permita a manipulação de conjuntos de qualquer tipo de elementos.

Desta forma, chegamos à identificação de duas classes:

- Conjunto, que modela uma abstração de conjuntos;
- Acervo, caso particular de conjunto, que modela o conceito de conjunto de livros de uma biblioteca.

Um acervo não acrescenta nenhum atributo ao conceito de conjunto. No entanto um acervo deve oferecer "formas" alternativas para consultar seus elementos (como consultar o acervo procurando livros escritos por um dado autor).

Usuário Os usuário da biblioteca são pessoas com autorização para tomarem emprestados livros do acervo da biblioteca. Cada abstração de usuário apresenta seus dados cadastrais e uma "ficha" relatando sua situação com a biblioteca (empréstimos que tem pendentes). O tempo de cada empréstimo e o número de livros que cada usuário pode ter emprestado depende do tipo de cada usuário, que pode ser enquadrado em uma dentre as seguintes categorias:

Usuários normais: são usuários sem privilégios excepcionais; podem ter emprestados, no máximo, dois livros, sendo que o tempo de empréstimo de cada livro é de uma semana.

Usuários estudantes: são usuários que podem retirar no máximo quatro livros por duas semanas de empréstimo cada um.

Usuários professores: são usuários que podem retirar até dez livros por um período de um mês para cada livro.

A necessidade de dados pessoais na caracterização de usuários possibilita a definição de usuário de biblioteca como especialização de pessoa. Desta forma, usando a classe Pessoa apresentada na página 35, estaremos reutilizando duas classes:

- Pessoa, que modela o conceito de pessoa através da manipulação de dados pessoais.

- Endereco, que descreve um atributo de Pessoa, utilizado para descrever o local onde reside a pessoa.

Resta ainda modelar o conceito de usuário de biblioteca. Para isto, consideraremos uma classe básica, que represente usuários genéricos, e três classes derivadas, modelando cada uma das categorias de usuários. Note que cada usuário da biblioteca deve ser enquadrado em exatamente uma das três categorias; por isto, a classe base da derivação deve ser abstrata.

Assim, completamos a modelagem de usuários com a criação das seguintes classes:

- Usuario, classe abstrata que abstrai características genéricas de usuários da biblioteca;
- UsuarioNormal, classe que representa a categoria de usuários da biblioteca sem nenhum privilégio adicional;
- UsuarioEstudante, classe que representa a categoria de usuários da biblioteca que são estudantes;
- UsuarioProfessor, representa a categoria de usuários de biblioteca que são professores.

Cadastro de Usuários O cadastro de usuários é o conjunto formado pelos usuários da biblioteca. Da mesma forma como ocorreu com a análise do conceito de acervo da biblioteca, o cadastro de usuários é um caso particular de conjunto.

Nesta hora, quando identificamos dois casos particulares de conjuntos, devemos tomar bastante cuidado com a modelagem de conjuntos para permitir que um mesmo modelo sirva para estas duas classes.

O importante desta análise é que encontramos mais um objeto:

- CadastroDeUsuários, que modela o conceito do cadastro de usuários da biblioteca[1].

Note ainda que este cadastro deve armazenar objetos de três classes já identificadas (UsuarioNormal, UsuarioProfessor e UsuarioEstudante). Por isto devemos caprichar no modelo de usuários e de cadastro para não criar problemas com a diversidade de elementos que o cadastro deverá manipular.

Empréstimo Empréstimo é o serviço principal oferecido pela biblioteca. Cada empréstimo relaciona um usuário com um livro (indicando que o livro foi retirado pelo usuário) e deve apresentar uma data indicando o término do período de validade do empréstimo. Uma solução para a modelagem deste serviço é caracterizá-lo como

[1] Na fase de implementação desta classe foi encontrado um problema sutil que fez com que este modelo obtido fosse "levemente" modificado. Entretanto, para fazer com que este estudo retrate um caso real de modelagem, apresentaremos o problema encontrado e sua solução no instante em que ele surgiu, ou seja, durante a implementação desta classe (feita na página 104).

um objeto que apresente como atributos as identificações do usuário e do livro relacionados e a data de devolução[2]. Encontramos, assim, a mais um objeto:

- Emprestimo, classe que representa o serviço de empréstimo de um livro a um usuário da biblioteca.

Conjunto O conceito de conjunto é bastante importante para diversas áreas da ciência da computação e uma modelagem satisfatória exigiria uma análise de domínio completa e totalmente fora do escopo deste trabalho. Por causa disto, optamos em descrever uma estrutura de dados básica e manipulá-la como um conjunto (mas não devemos esquecer que a solução mais recomendada para esta aplicação específica seria considerar um sistema gerenciador de base de dados).

Dentre tantas alternativas existentes para a representação de conjuntos optamos por sua modelagem através de uma lista ligada que, para sua implementação, deu origem a duas novas classes:

- No, classe que modela cada célula presente na lista ligada.
- ListaLigada, que modela o conceito de conjunto e tal que cada elemento esta armazenado em um de suas células.

Classe da Aplicação Além destas classes apresentadas, incluiremos uma nova para modelar o serviço oferecido pelo sistema, ou seja, vamos criar uma classe de aplicação. Na prática, esta classe corresponde ao serviço realizado pelo atendente da biblioteca: tem acesso ao cadastro de usuários e ao catálogo do acervo e é responsável pela execução dos serviços que a biblioteca oferece.

Como não modelamos os conjuntos desta aplicação por base de dados externa, esta classe de aplicação apresenta como atributos o cadastro de usuários da biblioteca e o acervo. A única função deste objeto é apresentar um menu de opções dos serviços oferecidos e executar a tarefa correspondente à opção selecionada.

Desta forma, chegamos ao último objeto necessário para a nossa aplicação:

- ServicoDeBiblioteca, que apresenta como únicas funções o construtor e o destrutor da classe.

5.4 Descrição dos Objetos

Agora que já mapeamos o sistema identificando os objetos necessários para seu funcionamento, vamos caracterizar os atributos e o comportamento apresentado por cada um deles. Para esta fase é recomendável descrever cada classe com o mesmo nível de detalhes

[2]A modelagem e implementação do conceito de datas foram feitas no capítulo anterior; por este motivo, ignoraremos sua modelagem, considerando-o como uma classe já implementada.

que usamos na apresentação da classe Data na página 68; no entanto, não o faremos para não tornar a leitura desta parte extensa e cansativa. Para retratar as classes e relacinamentos obtidos, aconselha-se a utilização de diagramas (como aqueles apresentados no capítulo anterior).

A seguir, exibiremos detalhes de cada classe identificada na fase anterior.

5.4.1 Livros Do Acervo

Para a modelagem dos livros do acervo da biblioteca, definimos duas classes relacionadas por herança: Livro, a classe base que representa livros genéricos, e LivroDeBiblioteca, classe derivadas que representa livros especificamente do acervo da biblioteca.

Descrição Das Classes

Livro: representa e manipula dados descrevendo livros. Cada livro é descrito pelos seguintes atributos

- título do livro;
- autor do livro;
- nome da editora;
- ano da edição.

Conforme mencionado, para cada livro é considerado apenas um autor. Esta classe apresenta funções para:

- inicializar objetos;
- copiar o valor de objetos desta classe;
- consultar o valor de cada atributo;
- alterar o valor cada um destes atributos;
- operações para entrada e saída.

LivroDeBiblioteca: representa e manipula livros de uma biblioteca. Além dos atributos herdados da classe Livro, objetos desta classe apresentam

- um código para o livro;
- referencias para sua ficha de empréstimos, caso exista.

Note que esta representação é simplificada, considerando-se as informações que constam para cada livro de uma biblioteca. Nos casos reais são também consideradas informações como data da compra do livro, livraria onde foi comprado, etc.

esta classe apresenta funções herdadas de sua classe base com a inclusão de funções específicas para:

- inicializar instâncias da classe;
- consultar e alterar seu código;
- para acusar seu empréstimo do item;
- acusar devolução do item;
- funções específicas para entrada e saída.

5.4.2 Usuários de Biblioteca

Na modelagem de usuários de biblioteca definida anteriormente, chegamos à seguinte hierarquia de classes:

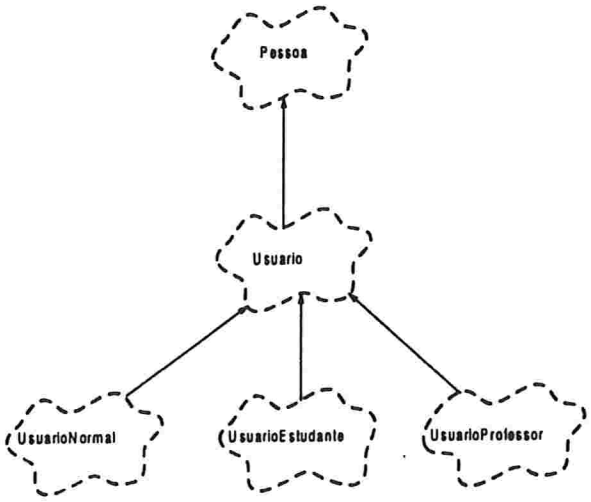

Figura 5.1: Hierarquia de classes para modelar o conceito de usuário de biblioteca

Descrição Das Classes

Pessoa: representa e manipula dados pessoais. Cada pessoa é descrita pelos atributos

- nome;
- endereco;
- telefone.

dos quais o atributo endereço é pertencente à classe Endereco. Esta classe apresenta funções membro para:

- inicializar objetos;
- copiar o valor de objetos desta classe;
- consultar o valor de cada atributo;
- alterar o valor cada um destes atributos;
- operações de entrada e saída.

Endereco: representa e manipula informações relacionadas com endereçamento. Cada objeto desta classe apresenta como atributos:

- nome da rua;
- cidade;
- estado;
- CEP.

Suas funções implementam os mesmos serviços relacionados para a classe Pessoa.

Usuario: é uma classe abstrata para modelar o conceito de usuário da biblioteca. O fato de ser abstrata implica que nunca existirá uma instância desta classe, ou seja, cada usuário deve obrigatoriamente ser classificado em uma dentre suas especializações. Esta é uma classe derivada de Pessoa que apresenta como atributos específicos:

- as informações referentes os empréstimos feitas pelo usuário.

Para descrever seu comportamento, esta classe apresenta funções para:

- inicializar objetos;
- copiar o valor de objetos desta classe;
- consultar o valor contido em cada atributo;
- alterar o valor cada um de seus atributos;
- operações de entrada e saída.

Observação: como esta é uma classe abstrata, muitas de suas funções são virtuais e devem obrigatoriamente ser implementadas nas suas classes derivadas.

UsuarioNormal, UsuarioEstudante e UsuarioProfessor: como esta três classes são praticamente idênticas, faremos seu tratamento conjuntamente.

UsuarioNormal: classe que implementa o comportamento de usuários que são autorizados a retirar no máximo 2 livros por um período de empréstimo de uma semana.

UsuarioEstudante: classes que implementa o comportamento de usuários que são autorizados a retirar no máximo 4 livros por um período de empréstimo de duas semana.

UsuarioProfessor: classe que implementa o comportamento de usuários que são autorizados a retirar no máximo 10 livros por um período de empréstimo de um mês.

Estas classes são derivadas de Usuario e apresentam como atributos específicos (para cada uma) duas informações:

- o número máximo de livros que o usuário pode retirar por empréstimo;
- o tempo de duração de cada empréstimo.

Note que estes atributos assumem o mesmo valor para todos os objetos de cada classe. Assim, devemos descobrir alguma forma para fazer com que estes atributos sejam associados a todas as instâncias das classes sem que cada uma destas instâncias precise conter uma cópia particular.

As funções destas classes implementam os seguintes serviços:

- inicializar instâncias da classe;
- consultar os valores de seus atributos específicos;
- (eventualmente) alterar o valor destes atributos;
- implementar operações de entrada e saída de objetos da classe.

5.4.3 Empréstimos

Para a representação dos empréstimos anotados pela biblioteca foi criado o objeto Emprestimo:

Emprestimo: representa e manipula empréstimos de livros aos usuários da biblioteca. Cada instância desta classe apresenta como atributo:

- código do usuário que solicitou o empréstimo;
- código do livro emprestado;
- data limite para a devolução do livro.

Esta classe apresenta funções para:

- inicializar objetos;
- copiar o valor de objetos desta classe;
- consultar o valor de cada atributo dos objetos da classe;
- alterar o valor cada um destes atributos;
- operações de entrada e saída.

5.4.4 Conjuntos e Suas Subclasses

Para a modelagem de conjuntos foi considerada uma estrutura de lista ligada. Caracterizamos duas classes para implementar o comportamento destas listas: No e ListaLigada. A partir destas classes, foram definidas duas classes derivadas para modelar o acervo da biblioteca e o cadastro de seus usuários.

No: descreve cada uma das células presentes em uma lista ligada através dos seguintes atributos:

- informação contida na célula;
- referencia para a próxima célula na lista.

Na modelagem destes elementos, devemos buscar uma implementação que seja independente da informação armazenada na célula. Desta forma, conseguiremos uma classe que, junto com ListaLigada poderá ser utilizada em outras aplicações nas quais sejam necessárias representações de conjuntos.

A classe No implementa as operações para:

- inicializar e destruir objetos;
- retornar uma referencia para o próxima célula da lista;
- fazer o apontador de próximo elemento apontar para uma outra célula;
- referenciar uma nova informação;
- retornar a informação armazenada;
- imprimir a informação armazenada na célula.

ListaLigada: descreve uma estrutura de dados de lista ligada na qual cada célula é alocada dinamicamente quando necessária. Apresenta três ponteiros como atributos:

- um para indicar o início da lista;
- outro que aponta para a última célula da lista;
- e o terceiro para indicar a posição corrente da lista.

Esta classe apresenta funções para:

- inicializar uma lista ligada como vazia;
- destruir uma lista (destruindo cada uma de suas células);
- inserir e remover elementos;
- funções específicas para iterar sobre a lista ligada:
  - fazer com que o elemento corrente seja o primeiro elemento da lista;
  - retornar a informação armazenada no elemento corrente;
  - fazer com que o elemento corrente seja o próximo elemento da lista;
  - verificar se a posição corrente corresponde ao final da lista ligada.

Acervo: classe derivada de ListaLigada que descreve um conjunto para o caso específico no qual este conjunto apresenta livros de biblioteca como elementos.

Os atributos desta classe são somente aqueles herdados de sua classe base, com a inclusão de novas funções específicas para realizar as seguintes tarefas:

- inicializar e destruir instâncias da classe;
- inserir um livro no acervo;
- remover um livro do acervo;
- fazer consultas a respeito dos livros presentes no acervo através das seguintes opções:
  - consulta pelo nome do autor;
  - consulta pelo título do livro;
  - consulta pelo código do livro no catálogo da biblioteca.
- carregar as informações cadastrais livros de um arquivo;
- gravar as informações cadastrais dos livros em um arquivo.

Cadastro: classe derivada de ListaLigada que descreve um conjunto no qual cada elemento apresenta dados cadastrais.

Na modelagem da classe cadastro, optamos pela descrição de uma classe genérica que sirva não somente para manipular um cadastro de usuários de biblioteca, mas também para outros tipos de dados cadastrais. Desta forma, descreveremos uma classe genérica que manipula objetos de qualquer classe desde que estes objetos apresentem uma informação que identifique o objeto (especificamente, um nome) além um código (que deve ser único, isto é, não aceitar duplicações).

Os atributos desta classe são aqueles mesmos herdados de ListaLigada e suas funções devem:

- inicializar e destruir instâncias;

- incluir e remover itens do cadastro;
- procurar um item pelo nome;
- procurar um item pelo código;
- ler o cadastro de um arquivo;
- gravar o cadastro em um arquivo.

5.4.5 Classe da Aplicação

ServicoDeBiblioteca: classe que representa os serviços oferecidos pela biblioteca.

Os atributos desta classe são:

- o cadastro de usuários da biblioteca;
- o acervo com os livros a disposição para empréstimo.

A única função membro pública que esta classe deve apresentar é seu construtor. O construtor apresentará um menu de opções de serviços na tela. A escolha de cada opção corresponde à chamada de uma função (privativa) específica para realizar a tarefa.

5.4.6 Implementação Das Classes

Agora que temos todas as informações necessárias, podemos implementar as classes que caracterizamos. Nesta fase devemos tomar os mesmos cuidados tomados com a classe Data no capítulo anterior

Um ponto em comum entre as classes obtidas é a opção pela utilização de alocação dinâmica de memória sempre que possível. Por este motivo, praticamente todas as classes do sistema irão apresentar as funções necessárias para controlar a semântica de cópia de objetos (o operador de atribuição e o construtor de cópia). Além disto, incluímos um construtor default para todas as classes consideradas.

Livro e LivroDeBiblioteca

Para a representação de livros, foram considerados quatro atributos:

- titulo: o título do livro (que é do tipo char*);
- autor: o nome do autor do livro (tipo char*);

- **editora**: o nome da editora responsável pela publicação (tipo char*);
- **anoDaEdicao**: ano da publicação do livro.

As funções desta classe são tão simples que a própria leitura do nome da função é suficiente para exprimir o funcionamento delas. A declaração desta classe está apresentada abaixo:

```
class Livro {
    char *titulo;
    char *autor;
    char *editora;
    int   anoDaEdicao;

public:
    Livro();
    Livro(char*, char*, char*, int);
    Livro(const Livro&);
    ~Livro();
    Livro &operator=(const Livro&);
    char *atribuiTitulo(char*);
    char *atribuiAutor(char*);
    char *atribuiEditora(char*);
    void  atribuiAnoDaEdicao(int);
    char  atribuiDados(char*, char*, char*, int);
    char* retornaAutor();
    char* retornaTitulo();
    int   retornaAnoDaEdicao();
    char* retornaEditora();
    void  leDados();
    friend ostream& operator<<(ostream &ost, const Livro &lvr);
    friend istream& operator>>(istream &ist, Livro &lvr);
};
```

Como classe derivada de Livro definimos uma classe que represente livros do acervo da biblioteca. Para a descrição destes objetos foram acrescentados (somente) dois atributos adicionais:

- **codigo**: o código que identifica o livro no acervo da biblioteca;
- **emprestadoA**: ponteiro para a ficha de empréstimo do livro; se este ponteiro valer NULL indica que o livro não está emprestado a nenhum usuário.

As funções membro desta classe também são simples, incluindo, além das funções básicas, funções para manipular o atributo codigo e efetuar o empréstimo do livro. A declaração da classe está apresentada abaixo:

```
class LivroDeBiblioteca: public Livro{
   char       *codigo;
   Emprestimo *emprestadoA;
public:
   LivroDeBiblioteca();
   LivroDeBiblioteca(char*, char*,  char*, char*, int);
   LivroDeBiblioteca(const LivroDeBiblioteca&);
   ~LivroDeBiblioteca();
   LivroDeBiblioteca& operator=(const LivroDeBiblioteca&);
   char empresta(Emprestimo*);
   void devolve();
   char* retornaCodigo() const;
   void  atribuiCodigo(char*);
   void leDados();
   friend ostream& operator<<(ostream &ost, const Livro &lvr);
   friend istream& operator>>(istream &ist, Livro &lvr);
};
```

Pessoa e Endereco

A implementação destas classes não apresentam detalhes particulares ou complicados que justificariam uma exposição detalhada.

```
class Endereco {
   char *rua, *cidade, *estado, *CEP;
public:
   Endereco();
   Endereco(char*, char*, char*, char*);
   Endereco(const Endereco&);
   ~Endereco();
   Endereco& operator=(const Endereco &outro);
   void  atribuiRua(char*);
   void  atribuiCidade(char*);
   void  atribuiEstado(char*);
   void  atribuiCEP(char*);
   void  atribuiEndereco(char*, char*, char*, char*);
   char* retornaRua() const;
   char* retornaCidade() const;
   char* retornaEstado() const;
```

```
   char* retornaCEP() const;
   void  leEndereco();
   friend istream& operator>>(istream &ist, Endereco &end);
   friend ostream& operator<<(ostream &ost, const Endereco &end);
};

class Pessoa {
   char     *nome, *telefone;
   Endereco  local;
public:
   Pessoa();
   Pessoa(char*, char*, Endereco);
   Pessoa( const Pessoa& );
   ~Pessoa();
   Pessoa& operator=(const Pessoa& outra);
   char*     retornaNome() const;
   char*     retornaTelefone() const;
   Endereco  retornaEndereco() const;
   void      atribuiNome(char*);
   void      atribuiTelefone(char*);
   void      atribuiEndereco(Endereco &end);
   void      lePessoa();
   friend istream& operator>>(istream&, Pessoa&);
   friend ostream& operator<<(ostream&, const Pessoa&);
};
```

Usuários Da Biblioteca

A implementação da classe UsuarioDeBiblioteca deve ser cuidadosa por trata-se da implementação de uma classe abstrata. Desta forma, devemos descrever aqui todas as funções que descrevem o comportamento de usuários de biblioteca, mas obrigando que algumas destas funções sejam implementadas nas classes derivadas (por dependerem de detalhes específicos que dever ser tratados nas subclasses).

Para a representação dos atributos desta classe, estão declaradas três variáveis:

- codigo: codigo que identifica o usuário na biblioteca (o sistema recusará o cadastramento de usuários com códigos duplicados);
- temEmprestado: usando o tipo conjunto que devemos implementar, este atributo será declarado como um conjunto de empréstimos;
- numeroDeEmprestimos: uma variável inteira que contém o número de livros que o usuário tem emprestado.

A declaração da classe está apresentada abaixo:

```
class UsuarioDeBiblioteca: public Pessoa{
    char* codigo;
    ConjEmprestimos temEmprestado;
    int  numeroDeEmprestimos;
public:
    UsuarioDeBiblioteca();
    UsuarioDeBiblioteca(UsuarioDeBiblioteca &);
    virtual ~UsuarioDeBiblioteca();
    UsuarioDeBiblioteca operator=(UsuarioDeBiblioteca &);
    char anotaEmprestimo(Emprestimo*);
    void anotaDevolucao(Emprestimo*);
    char podeTerEmprestimo();
    char* retornaCodigo();
    virtual void leAtributos()= 0;
    virtual int  retornaMaximoDeEmprestimos()= 0;
    virtual Data retornaTempoDeEmprestimo()= 0;
    virtual UsuarioDeBiblioteca* copia()= 0;
    virtual void leUsuario(istream&)= 0;
    virtual void escreveUsuario(ostream&) = 0;
};
```

As funções declaradas nesta classe pode ser dividida em dois grupos:

- funções genéricas, comuns para todo tipo de usuário que, por este motivo, estão implementadas na própria classe para as classes derivadas herdarem;
- funções que dependem do tipo específico de usuário que, por este motivo, estão declaradas como virtuais para serem implementadas nas classes derivadas.

  A notação $= 0$ na declaração destas funções virtuais desta classe denota que estas funções não serão implementadas nesta classe mas devem obrigatoriamente ser implementadas nas classes derivadas. Além disto, esta declaração também implica que nunca, de forma alguma, poderão existir instâncias desta classe.

UsuarioNormal, UsuarioProfessor e UsuarioEstudante

A finalidade destas classes, todas derivadas da classe UsuarioDeBiblioteca, no sistema é particularizar a forma de uso da biblioteca pelas suas categorias de usuários. Como estas três classes são idênticas, vamos tratá-las conjuntamente.

A distinção entre estes usuários é o tempo de empréstimo e o número de itens que podem retirar emprestados. Para modelar este comportamento foram incluídos na descrição destes objetos mais dois atributos:

- tempoDeEmprestimo: que descreve o período de empréstimo associado a cada categoria de usuários da biblioteca;
- maximoDeItensEmprestados: que descreve o número máximo de livros que os usuários de cada categoria podem retirar emprestados.

Como estes atributos deve assumir o mesmo valor para todas as instâncias de cada uma destas três classes, eles foram definidos como membros estáticos da classe.

Para a manipulação destes objetos, estão incluídas as funções declaradas na classe base como virtuais e funções estáticas para, eventualmente, alterar o valor dos membros estáticos. As declarações destas três classes são apresentada a seguir:

```
class UsuarioNormal: public UsuarioDeBiblioteca{
    static int  tempoDeEmprestimo;
    static int  maximoDeItensEmprestados;
public:
    UsuarioNormal();
    UsuarioNormal(const UsuarioNormal &);
    int     retornaMaximoDeEmprestimos();
    int     retornaTempoDeEmprestimo();
    virtual UsuarioDeBiblioteca* copia();
    void leAtributos();
    void escreveUsuario(ostream& ost);
    void leUsuario(istream& ist);
    static  atribuiTempoDeEmprestimo(int tempo);
    static  atribuiMaximoDeItensEmprestados(int num);
};


class UsuarioEstudante: public UsuarioDeBiblioteca{
    static Data  tempoDeEmprestimo;
    static int   maximoDeItensEmprestados;
public:
    UsuarioEstudante();
    UsuarioEstudante(const UsuarioEstudante &);
    int  retornaMaximoDeEmprestimos();
    Data retornaTempoDeEmprestimo();
    static atribuiTempoDeEmprestimo(Data tempo);
```

```
    static atribuiMaximoDeItensEmprestados(int num);
    virtual UsuarioDeBiblioteca* copia();
    void leAtributos();
    void escreveUsuario(ostream& ost);
    void leUsuario(istream& ist);
    friend istream& operator>>(istream& ist, UsuarioEstudante& usu);
    friend  ostream& operator<<(ostream& ost, const UsuarioEstudante& usu);
};


class UsuarioProfessor: public UsuarioDeBiblioteca{
    static Data  tempoDeEmprestimo;
    static int   maximoDeItensEmprestados;
public:
    UsuarioProfessor();
    UsuarioProfessor(const UsuarioProfessor &);
    int  retornaMaximoDeEmprestimos();
    Data retornaTempoDeEmprestimo();
    static atribuiTempoDeEmprestimo(Data tempo);
    static atribuiMaximoDeItensEmprestados(int num);
    virtual UsuarioDeBiblioteca* copia();
    void leAtributos();
    void escreveUsuario(ostream& ost);
    void leUsuario(istream& ist);
    friend istream& operator>>(istream& ist, UsuarioProfessor& usu);
    friend ostream& operator<<(ostream& ost, const UsuarioProfessor& usu);
};
```

5.4.7 Empréstimos

Para modelar este serviço foi criada a classe Emprestimo, cuja descrição é caracterizada por três atributos:

- codUsuario: o código do usuário que retirou o livro emprestado;
- codLivro: o código do livro que foi emprestado;
- dataDevolução: atributo da classe Data que contém a data limite para a devolução do livro.

Como funções membro, esta classe apresenta apenas aquelas necessárias para a consulta do valor de seus atributos e para alteração destes valores:

```
class Emprestimo{
    char *codUsuario;
    char *codLivro;
    Data  dataDevolucao;
public:
    Emprestimo();
    Emprestimo(const Emprestimo &);
    Emprestimo(char*, char*, Data);
    Emprestimo& operator=(const Emprestimo &);
    void  atribuiCodUsuario(char*);
    void  atribuiCodLivro(char*);
    void  atribuiDataDeDevolucao(Data);
    void  atribuiEmprestimo(char*, char*, Data);
    char* retornaCodUsuario();
    char* retornaCodLivro();
    Data  retornaDataDevolucao();
    friend istream& operator>>(istream &, Emprestimo &);
    friend ostream& operator<<(ostream &, Emprestimo &);
};
```

5.4.8 Conjuntos e Suas Subclasses

No e ListaLigada

Para obter classes genéricas que representem conjuntos, usamos templates na implementação de listas ligadas. Desta forma, uma declaração como

```
ListaLigada<LivroDeBiblioteca> umConjuntoDeLivros;
```

é suficiente para declara um conjunto tal que seus elementos são da classe LivroDeBiblioteca

A template ListaLigada recebe como parâmetro o tipo da informação que será armazenada e usa este tipo para declarar o tipo de células que serão as componentes da lista. Abaixo está apresentada a declaração desta template:

```
template <class TipoBase> class ListaLigada {
   No<TipoBase> *inicio;
   No<TipoBase> *ultimo;
   No<TipoBase> *corrente;
public:
   ListaLigada();   // construtor inicializa uma lista vazia
```

```
   ListaLigada(ListaLigada<TipoBase>&);
   ~ListaLigada();
   ListaLigada<TipoBase>& operator=(ListaLigada<TipoBase>&);
   void          insere(No<TipoBase>*);
   No<TipoBase>* remove(No<TipoBase>*);
   void          posicionaNoInicio();
   char          fimDaListaLigada() const;
   TipoBase*     elementoCorrente() const;
   No<TipoBase>* noCorrente() const;
   No<TipoBase>* proximo();
};
```

Um detalhe importante para a utilização desta classe é que para inserir um elemento na lista é necessário inserir um objeto no que contenha o elemento. Este elemento é desalocado quando for executado o destrutor de ListaLigada.

Por sua vez, a template no recebe como parâmetro o tipo da informação que será armazenada na lista ligada (cujo tipo é indicado por Info).

```
template <class Info> class No {
   Info *elem;
   No   *prox;
public:
   No();
   ~No();
   No* proximo();
   void insere(No *n);
   void insereInfo(Info* umaInfo);
   Info* retornaInfo() const;
   void imprimeInfo(ostream& ost) const;
};
```

Cadastro Dos Usuários

Para a implementação do cadastro de usuários foi considerada a hipótese de se implementar uma template que funcionasse para tratar de dados cadastrais genéricos. Exigiríamos somente que o tipo base usado como parâmetro da template contivesse uma função chamada retornaNome() e outra chamada retornaCodigo(). Pensou-se que uma única classe genérica fosse suficiente para implementar o cadastro de usuários do sistema... mas encontramos um problema:

> Quando tentamos implementar uma função para ler o cadastro de um arquivo verificamos que deveria ser previsto quais os tipos dos dados que iriam

ser lidos. Assim, leríamos as informações de um usuário e, dependendo de sua categoria, alocaríamos dinamicamente memória para seu armazenamento. Ou seja, seria necessário escrever as linhas como

```
entra >> tipoDeUsuario;
switch (tipoDeUsuario) {
   case 0: umUsuario= new UsuarioNormal; break;
   case 1: umUsuario= new usuarioEstudante; break;
   case 2: umUsuario= new usuarioProfessor; break;
   default: return 0;
}
```

na template Cadastro que faria com que perdesse sua generalidade.

Para solucionar este problema, criamos uma nova classe, derivada da template Cadastro, que implementa este serviço de entrada e saída de dados cadastrais para usuários de biblioteca.

Abaixo está apresentada as classes Cadastro e CadastroDeUsuarios, que acabamos de criar:

```
template <class Item> class Cadastro: public ListaLigada<Item> {
public:
    Cadastro();
    Cadastro(Cadastro<Item> &);
    ~Cadastro();
    Cadastro<Item>& operator=(Cadastro<Item> &);
    Cadastro<Item> procuraNome(char *);
    Item*          procuraCodigo(char *);
    void  incluiItem(Item*);
    Item* removeItem(char*);
};


class CadastroDeUsuarios: public Cadastro<UsuarioDeBiblioteca> {
public:
    char  leCadastroDeArquivo(char*);
    char  gravaCadastroEmArquivo(char*);
};
```

Acervo

A classe acervo é bastante similar à template cadastro, com a exceção de que esta classe implementa um conjunto especificamente de livros do acervo de uma biblioteca.

```
class Acervo: public ListaLigada<LivroDeBiblioteca> {
public:
    Acervo();
    Acervo(Acervo&);
    ~Acervo();
    Acervo& operator=(Acervo&);
    void  incluiLivro(LivroDeBiblioteca &);
    void  consultaAAcervo();
    void  removeLivro(char *);
    char  leAcervoDeArquivo(char*);
    char  gravaAcervoEmArquivo(char*);
    Acervo procuraAutor(char *);
    Acervo procuraTitulo(char *);
    Acervo procuraCodigo(char *);
};
```

5.4.9 Serviço da Biblioteca

A última classe do sistema, ServicoDeBiblioteca, é a classe da aplicação que coordena os serviços da biblioteca:

```
class ServicoDeBiblioteca {
    Acervo              umAcervo;
    CadastroDeUsuarios  umCadastro;
    int  menu(char**);
    void emprestimoDeLivro(Acervo &);
    void cadastramentoDeLivro(Acervo &);
    void cadastramentoDeUsuarios(CadastroDeUsuarios &);
    void consultaAAcervo();
public:
    ServicoDeBiblioteca();
    ~ServicoDeBiblioteca();
};
```

Com este modelo feito, a execução do sistema corresponde à simples declaração de uma variável desta classe:

```
int UsuarioNormal::tempoDeEmprestimo= Data(7,0,0);
int UsuarioNormal::maximoDeItensEmprestados= 2;
int UsuarioEstudante::tempoDeEmprestimo= Data(14,0,0);
int UsuarioEstudante::maximoDeItensEmprestados= 4;
int UsuarioProfessor::tempoDeEmprestimo= Data(0,1,0);
int UsuarioProfessor::maximoDeItensEmprestados= 10;

main()
{ ServicoDeBiblioteca s; }
```

5.5 Análise do Resultado

Para a solução deste problema foram criadas as seguintes classes:

- Data
- Pessoa
- Endereco
- UsuarioDeBiblioteca
- UsuarioNormal
- UsuarioEstudante
- UsuarioProfessor
- Livro
- LivroDeBiblioteca
- Emprestimo
- No
- ListaLigada
- Acervo
- Cadastro
- CadastroDeUsuarios
- ServicodeBiblioteca

Destas classes, Data, Pessoa, Endereco, Livro, No, ListaLigada e Cadastro são independentes do domínio do problema e podem ser reaproveitadas em outras aplicações.

Como esta aplicação foi desenvolvida sem a utilização de classes previamente prontas, o custo total do projeto tornou-se relativamente elevado. Este ponto desfavorável será compensado em aplicações futuras que utilizarem algumas destas classes como parte de seu projeto. Claramente, para se reutilizar alguma classe, essa classe deve ter sido implementada anteriormente.

Devido à arquitetura baseada em objetos deste sistema e da natureza do ciclo de vida para ele, torna-se difícil particionar o custo total do desenvolvimento entre cada uma das atividades envolvidas. Notadamente, grande parte do custo está associado às fases iniciais do projeto de cada classe (análise, desenvolvimento e implementação). Após implementada uma classe, o trabalho envolvido na sua manutenção torna-se bastante reduzido e, a longo prazo, o custo do desenvolvimento passa a ser reduzido (comparando-se com as técnicas estruturadas).

Além disso, merecem destaque os seguintes detalhes:

- Um aspecto que parece negativo na modelagem conceitual com objetos é que mesmo tipos de dados simples (como a classe Data) exige uma interface completa. Por causa disto, escreve-se bastante nas fases iniciais do projeto.
- Para a implementação deste sistema foi de fundamental importância um ambiente de edição baseados em janelas devido ao número elevado de módulos componentes do sistema.
- Implementação de classes individuais facilita na depuração de erros.

Este sistema implementado corresponde a um primeiro protótipo cujo funcionamento pode ser completado com as seguintes melhorias:

- implementar o serviço de tratamento de reservas de livros;
- estabelecer uma melhor interface com o usuário do sistema;
- fazer tratamento de livros escritos por mais de um autor;
- oferecer ao usuário melhores recursos para consulta pelo nome do autor, palavras chave, etc.

Resta ainda enfatizar a natureza do exemplo apresentado neste capítulo, apresentado com o objetivo de ilustrar, através de uma aplicação comercial, a viabilidade da utilização de técnicas de DOO na produção de programas de computador.

Para a escolha da aplicação, o gerenciamento do serviço de empréstimos de livros do acervo de uma biblioteca, foi considerado o fato de este tipo de problema ser bastante comum em programação comercial e a conseqüente facilidade de se comparar os métodos orientados a objetos com os métodos tradicionais de programação.

No entanto, é importante destacar que para o exemplo específico apresentado aqui, e para a grande maioria dos problemas freqüentes em programação comercial, a utilização de um sistema gerenciador de bancos de dados seria mais recomendável uma vez que a descrição e a manipulação de dados é bastante simplificada nestes sistemas. Por outro lado, a escolha de um exemplo diferente (como um problema de simulação, por exemplo), poderia ser mais recomendável para ilustrar a viabilidade comercial da utilização de técnicas de programação orientada a objetos mas correria o risco de tornar-se específica demais para apresentar DOO fora do meio científico.

Por este motivo, este projeto isolado pode dar a impressão de que programação orientada a objetos não é viável, uma vez que um sistema gerenciador de base de dados é capaz de fornecer um resultado semelhante a um custo muito mais reduzido. Para evitar este tipo de mal entendido, é importante realçar que:

- o exemplo foi desenvolvido com finalidade didática, para expor as técnicas de desenvolvimento orientado a objetos apresentadas no capítulo anterior;

- apesar de este projeto ter um custo relativamente elevado, a elaboração futura de aplicações semelhantes será bastante simplificada uma vez que será possível reaproveitar partes deste projeto;

- este tipo de implementação pode tornar-se viável em aplicações mais complexas que envolvam manipulação de bases de dados acopladas a operações como manipulação de matrizes, etc.

Cabe realçar que neste trabalho foi buscado apresentar idéias baseadas na tecnologia de orientação a objetos no desenvolvimento de programas, dando destaque especial à modelagem de dados (objetos), sem dar destaque excessivo para a linguagem de programação. Foi buscado, desta forma, mostrar que DOO trata-se de uma forma de modelagem de sistemas através de objetos. A escolha da linguagem de programação utilizada, C++, foi considerada devido à grande popularidade de C e o crescente interesse que C++ vem despertando nos profissionais da área de programação de sistemas. Dentre as características de C++ que motivou sua escolha para o projeto, destaca-se:

- sua semelhança com a linguagem C, que facilita sua apresentação e aprendizado;
- por tratar-se de uma linguagem de domínio geral, C++ pode ser utilizada em diversas áreas de aplicação (e não somente no gerenciamento de dados);
- disponibilidade de compiladores para C++ para a maioria das máquinas existentes;
- portabilidade dos programas escritos em C++.

Somente com o tempo, com o desenvolvimento de outras aplicações, com a evolução e manutenção deste sistema, poderemos realmente comprovar a validade de tudo que tem sido publicado referente a este novo modelo de programação.

Capítulo 6

Um Final?

Neste trabalho abordamos a aplicação de conceitos relacionados com o paradigma de programação orientada a objetos no desenvolvimento de programas.

Esta apresentação teve inicio propriamente no capítulo 2 quando estudamos o problema de desenvolvimento de programas dando ênfase às duas abordagens mais comuns, a mais antiga e tradicional, baseada nas ações que o computador deve executar, e a mais recente, a partir da qual se originou o modelo de POO, baseada na descrição de dados. Na introdução deste capítulo foi apresentada uma breve introdução histórica e os dois métodos acima descritos foram ilustrados com breves exemplos de desenvolvimento. Para concluir, foi apresentado o conceito de ciclo de vida no desenvolvimento de um software.

A partir do capítulo seguinte, restringimos nossa abordagem para o modelo de POO. Inicialmente foi apresentada uma introdução ao paradigma com as definições de seus conceitos principais, dos quais destacamos modularidade, encapsulamento, abstração, herança e polimorfismo. No encerramento, foram estudados os recursos que as linguagens OO oferecem para possibilitar o reaproveitamento de partes de um programa em outras aplicações.

No capítulo 4, usamos aqueles conceitos expostos no capítulo anterior para, finalmente, desenvolver programas. Para isto, foram consideradas atividades fundamentais no processo de programação baseado em objetos, considerando da identificação das classes (que corresponde, principalmente, à análise do sistema) até sua implementação. Terminamos com a exposição de sugestões de novos modelos de ciclo de vida para sistemas produzidos com objetos.

No capítulo seguinte foram usadas as idéias de DOO para resolver um problema real. Apesar de termos restringido o problema proposto, foi obtido um protótipo que pode facilmente ser adaptado para resolver o problema completo.

6.1 Um Comentário Final...

Uma grande lição aprendida com elaboração deste trabalho foi a importância do conceito de abstração no desenvolvimento de sistemas. POO, na verdade, enfatizou a importância na relação que os programas devem ter com o mundo real, universo onde ocorrem os problemas que devemos resolver com o computador.

Uma vez que existe um abismo entre os problemas e os programas que se propõem a resolver estes problemas, POO implementa uma forma elegante de se descrever os problemas (através da caracterização das entidades envolvidas no domínio do problema) e obter a solução (deixando que as próprias entidades envolvidas no problema "se resolvam"). Conseguiu-se, assim, um modelo de programação cujos produtos básicos (as classes) contêm informações expressivas que facilitam na administração do ciclo de vida do sistema.

É importante destacar que POO pressupõe um método, que não adianta adotar uma linguagem orientada a objetos e esperar que nossos problemas se resolvam imediatamente. POO é "apenas" um bom método de programação; e a nossa sorte é que existem também linguagens de programação que dão suporte a todos os aspectos relacionados com este modelo.

Na primeira aula do curso de programação orientada a objetos, ministrado no Instituto de Matemática e Estatística da Universidade de São Paulo, antes de apresentar o curso aos alunos, apliquei o seguinte questionário:

- Considerando as necessidades de sua função profissional (como programador) e seu atual estágio profissional, você se considera:
  ( ) principiante
  ( ) experiente
  ( ) não sabe dizer
- Você está satisfeito com seu desempenho profissional? (Justifique)
- Na sua opinião, os programas que escreve são:
  ( ) bons
  ( ) satisfatórios
  ( ) ruins
  ( ) não sabe dizer
  Justifique sua resposta.
- O que é um bom programa?

- Você saberia descrever o método que utiliza para, a partir da descrição de um problema que você deve resolver, produzir um programa para solucioná-lo?

  Se sua resposta for afirmativa, descreva seu método; caso contrário, justifique sua resposta.

A proposta deste questionário foi de motivar uma auto-análise por parte dos alunos para, posteriormente, apresentar POO não como uma revolução mas como um método que exige organização e disciplina. Quando alguém levanta a possibilidade de mudar de método de programação para POO, na verdade esta pessoa questionando a eficácia da forma com que programa.

Das duas, uma: Será que o método que esta pessoa utiliza até então é ruim ou é a pessoa que não tem competência para seguir o método? Ou será que não existe método?

Como caracterização de bom programa, muitos alunos usaram o termo "bem estruturado". Daí surgiu-me a pergunta: será que a partir de agora estas pessoas vão usar o termo "bem orientado a objetos" para descrever um bom programa?

Bibliografia

[Birt73] Birtwistle, G. M., Dahl, O-J., Myhrhaug, B., Nygaard, K. *Simula Begin.* Auerbach , Philadelphia, 1973.

[Booch82] Booch, G. *Object-oriented design.* Ada Letters, 3 (1), p.64-76, mar-abr.1982.

[Booch91] Booch, G. *Object-oriented design with applications.* Benjamin/Cummings, 1991.

[Byte81] Byte. McGraw Hill, v.6, n.8, ago.1981.

[Casa87] Casanova, M.A.,Giorno, F.A.C., Furtado, A.L. *Programação em Lógica e a Linguagem PROLOG.* Edgard Blucher,São Paulo, 1987.

[Dijks72] Dijkstra, E. W. *Notes on structured Programmming.* In. Dahl et. al. Structured programming. Academic Press, New York, 1972.

[Ellis90] Ellis, M. A., Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Fowler93] Fowler, M. *OO methods.* Object-oriented analysis and design. Sigs Publications, 1993.

[Gibbs90] Gibbs, S. et. al. *Class management for software comunities.* CACM, v.33, n.9, pag.90-103, set.1990.

[Hender90] Hendeson-Sellers, B., Edwards, J.M. *The object-oriented systems life cicle.* CACM, v.33, n.9, pag.143-159, set.1990.

[Kors90] Korson, T., McGregor, J. D. *Understanding object-oriented: a unifying paradigm.* CACM, v.33, n.9, pag.40-60, set.1990.

[Krue92] Krueger, C. W. *Software reuse.* ACM Computing Surveys, 2 (24), p.131-183, jun.1992.

[Mart91] Martinez, R. *Programación en reglas de Producción.* Edição EBAI, Rio de Janeiro, 1991.

[Meira88] Meira, S. R. L. *Introdução Programação Funcinal.* Editora da UNICAMP, Campinas, 1988.

[Mona92] Monarch, D. E., Puhl, G. I. *A research typology for object-oriented analysis and design.* CACM 9 (18), p.35-47, set.1992.

[Mullin89] Mullin, M. *Object Oriented Program Design.* Addison-Wesley, 1989.

[Nier89] Nierstrasz, O. *A survey of object-oriented concepts.* In Object-oriented concepts, databases and applications, ed W. Kim and F. Lochovsky. ACM Press ans Addison-Wesley, p.3-21, 1989.

[Nyga86] Nygaard, K. *Basic concepts in object oriented programming.* SIGPLAN Notices, v.21, n.10, p.128-132, out.1986.

[Odell92] Odell, James. *More than a programming language.* Object Magzine, 2(4), nov-dez.1992, pag.47-49.

[Pasc86] Pascoe, G.A. *Elements of object-oriented programming.* Byte, v.11, n.8, p.139-144, ago.1986.

[Quine89] Quine, W.V. *Relatividade ontológica e outros ensaios.* In. Ryle, G. et. al. *Ensaios* (Coleção Os Pensadores) Nova Cultural, São Paulo, 1989.

[Shlaer92] Shlaer, S., Mellor, S. J. *Plunging into object-oriented terminology.* Object Magazine, v.2, n.4, pag 39-41, nov.1992.

[Stro83] Stroustrup, B. *Adding classes to the C language: an exercise in language evolution.* Software - Practice and Experience, v.13, n.2, p.139-161, 1983.

[Stro88] Stroustrup, B. *What is Object-Oriented Programming?* IEEE Software, v.3, n.5, mai.1988.

[Stro93] Stroustrup, B. *We're not in Kansas anymore.* Object-oriented analysis and design. Sigs Publications.

[Taka90] Takahashi, T., Liesenberg, H. K. E. *Programação Orientada a Objetos.* IME-USP, São Paulo, 1990.

[Thom89] Thomas, D. *What's an object?* Byte, v.14, n.3, p.231-240, mar.1989.

[Veloso87] Veloso, P. A. S. *Estruturação e verificação de programas com tipos de dados.* Edgard Blucher, São Paulo, 1987.

[Vidart88] Vidart, J., Tasistro, A. *Programación lógica y funcional.* Edição EBAI, 1988.

[Wegner76] Wegner,P. *Programming languages - the first 25 years.* IEEE Trans. on Computers, v.25, n.12, dez.1976.

[Wegner89] Wegner, P. *Learning the language.* Byte, v.14, n.3, p.245-253, mar.1989.

[Wirfs90] Wirfs-Brock, R. J., Johnson, R. *Surveying current research in object-oriented design.* CACM, v.33, n.9. p.105-124, set.1990.

[Wirth71] Wirth, N. *Program development by stepwise refinement.* CACM, 4 (14), abr.1971, p.221-227.