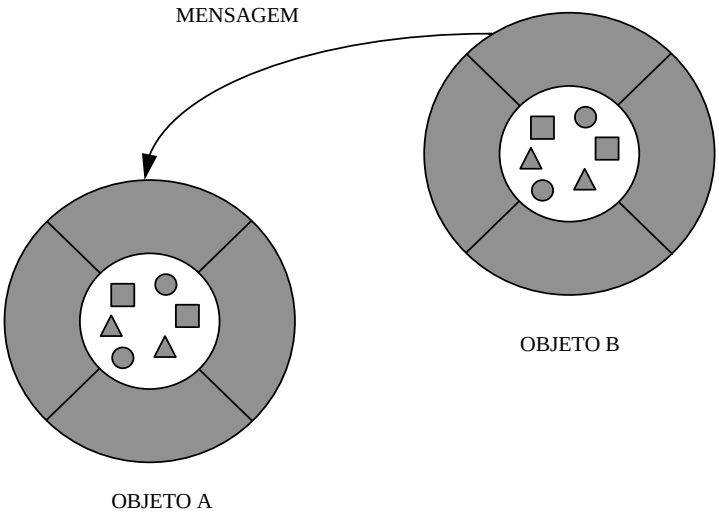
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DA PARAÍBA
DIRETORIA DE ENSINO – COORDENAÇÃO DE INFORMÁTICA

INTRODUÇÃO AO

PARADIGMA DE

ORIENTAÇÃO A OBJETOS



AUTOR: PROF. CARLOS ALBERTO KAMIENSKI

JOÃO PESSOA, JULHO DE 1996

ÍNDICE

1. Introdução **1**
1.1. Por que estudar o paradigma de orientação a objetos ? 1
1.2. A natureza das novas aplicações 1
1.3. O desafio das novas tecnologias 2
1.4. Os ambientes de programação modernos 2
1.5. Programação Orientada a Objetos 3
2. Conceitos básicos e terminologia **5**
2.1. Abstração de dados 5
2.2. Objetos 6
2.3. Mensagens 8
2.4. Métodos 9
2.5. Classes 9
2.6. Herança 11
2.6. Polimorfismo 13
2.7. Relações entre Objeto, Classe e Herança 15
2.7.1. Classificação/Instanciação 15
2.7.2. Generalização/Especialização 15
2.7.3. Composição/Decomposição 16
3. Avaliação da POO **16**
3.1. Vantagens da POO 17
3.2. Desvantagens da POO 18
3.3. O estilo de programação com POO 19
3.4. Programação Dirigida por Eventos 20
4. Linguagens Típicas **21**
4.1. A linguagem de programação SmallTalk 22
4.2. A linguagem de programação C++ 25
4.3. A linguagem de programação Object Pascal 27
5. O paradigma de orientação a objetos **31**
5.1. Desenvolvimento Orientado a Objetos 31
5.2. Análise Orientada a Objetos 33
5.3. Projeto Orientado a Objetos 33
5.4. Técnica de Modelagem de Objetos 34
5.4.1. Análise 35
5.4.2. Projeto do sistema 36
5.4.3. Projeto dos Objetos 36
5.5. Bancos de Dados Orientados a Objetos 37
5.6. Outras utilizações da Orientação a Objetos 38
5.6.1. Reengenharia de Processos Orientada a Objetos 38
5.6.2 Ambientes de Desenvolvimento de Software Orientados a Objetos 38
5.6.3 Interfaces Gráficas de Usuário Orientadas a Objetos 39
5.6.4 Sistemas Operacionais Orientados a Objetos 39
6. Bibliografia **39**

1. Introdução

O paradigma de Orientação a Objetos (OO) não é apenas um nome pomposo, uma nova terminologia ou modismo sem efeito prático. Possui o objetivo, e consegue obtê-lo em conjunto com outras tecnologias, de atenuar o que se convencionou chamar de "crise do software". No início do era dos computadores o hardware era tão caro que o custo do software era muitas vezes desprezado nas organizações. As aplicações eram também, se comparadas com a atualidade, representativamente mais simples. Com o passar do tempo observou-se que os custos associados ao hardware diminuíram gradativamente, enquanto que os custos do software aumentaram assustadoramente. Hoje, em uma instalação os custos associados com o software superam em várias vezes cos custos de hardware, e a tendência é que isto aumente cada vez mais.

1.1. Por que estudar o paradigma de orientação a objetos ?

Existem qualidades que atualmente todas as empresas querem atribuir a seus produtos, que nem implementam corretamente. Três qualidades mais notórias são:

1. **Sistema aberto**: tudo atualmente é "aberto", mesmo que não siga padrões (que define os sistemas abertos) de mercado ou de direito.
1. **Arquitetura cliente/servidor**: tudo atualmente é "cliente/servidor, mesmo que não possibilite a divisão do software entre clientes e servidores.
1. **Orientação a objetos**: tudo atualmente é orientado a objetos, mesmo que não implemente objetos, classes e herança.

O primeiro bom motivo para estudar o paradigma de orientação a objetos é distinguir quais produtos são orientados a objetos e quais não são. O segundo é que a orientação a objetos não é só moda, está-se utilizando e vai se utilizar cada vez mais desde Reengenharia de Processos Orientada a Objetos até Bancos de Dados Orientados a Objetos, passando pela análise, projeto e programação.

1.2. A natureza das novas aplicações

A natureza das novas aplicações de computadores tem evoluído mais rapidamente que a capacidade de concretizá-las. Algumas características críticas de novas aplicações e necessidades emergentes incluem:

- *Tamanho e complexidade*: Software de uso geral tem dezenas de Mbytes de código-fonte. Mesmo "pacotes" utilitários cuja arquitetura interna passa em larga medida despercebida pelo usuário comum chegam facilmente a essa faixa. Por outro lado, a própria arquitetura interna das aplicações é altamente complexa, incluindo paralelismo, transações múltiplas, forte interação gráfica, etc.
- *Alta confiabilidade*: Na medida em que sistemas automatizados vão sendo introduzidos para assistir usuários comuns não especializados em tarefas

complexas, a margem de erro aceitável se torna cada vez menor, uma vez que, a rigor, o próprio operador de um sistema está a reboque do sistema e não o contrário.

- *Alto desempenho*: Novas aplicações demandam alto desempenho em cálculos (ex.: rastreamento por satélites, previsão do tempo, sistemas de informações geográficas, CAD, etc.) e/ou em coordenação e comunicação entre as partes (usando redes de computadores, por exemplo).

1.3. O desafio das novas tecnologias

O software é sempre muito caro, os prazos não são cumpridos e a manutenção é problemática. O problema básico a ser enfrentado em desenvolvimento de software é, em poucas palavras:

CONSTRUIR SOFTWARE RÁPIDO, BARATO E FLEXÍVEL

- **Construir Rápido**: Na medida em que os requisitos e plataformas são fluídos e evolutivos, despender tempo excessivo é terrível. Por outro lado, com o "backlog" de aplicações a desenvolver cada vez maior, a construção de software mais rápido é uma necessidades na prática.
- **Construir Barato**: O custo de desenvolvimento é relativamente alto, devido à estruturação da tarefa nos moldes atuais em "linhas de montagem" hiper-dimensionadas, e devido à baixa produtividade. É necessário um aumento drástico na quantidade e qualidade de software produzido por indivíduos, passando pela reutilização de software já desenvolvido.
- **Construir Flexível**: Sistemas devem ser arquitetados de tal forma que sua alteração para qualquer fim (modificação ou extensão para atender a novos requisitos, reutilização em outros projetos, etc.) seja fácil.

Linhas de ação para atacar esses problemas têm incluído:

- investigação de novas técnicas e seu suporte adequado em ambientes integrados para desenvolvimento de software;
- revisão do modelo clássico de ciclo de vida de sistemas e de métodos de desenvolvimento de software;
- melhor compreensão de aspectos fundamentais envolvidos no processo de software.

Tais linhas de ação têm em geral seguido caminhos isolados, quando não conflitantes. Há, contudo, uma ênfase crescente em idéias convergindo para o chamado **paradigma de orientação a objetos**, como uma possível foco integrador dessas linhas.

1.4. Os ambientes de programação modernos

Os ambientes de Desenvolvimento de Software têm evoluído muito nos últimos anos para incorporar várias tecnologias num movimento sinérgico em direção obter melhores condições do desenvolvimento de sistemas. Com relação às ferramentas específicas para a programação, atualmente várias características têm se firmado em termos de mercado:

- *Programação visual*: programação visual se refere à utilização de interfaces gráficas interativas para o desenvolvimento de programas. Dessa forma, o programador no desempenho da sua tarefa, obtêm os mesmos benefícios das GUI (Grafical User Interface - interface gráfica de usuário) que ele projeta.
- *Programação dirigida por eventos*. Nos ambientes gráficos o processamento perde o caráter puramente seqüencial que se está acostumado para ser dirigido pelos eventos gerados pelo usuário (seção 3.4).
- *Programação para Windows*: existem vários ambiente que utilizam programação visual e processamento dirigido por eventos, mas o Windows tornou-se o padrão em termos de ambiente "desktop". A programação para Windows têm certas características específicas, como o tratamento de mensagens e recursos, que no entanto, somente são de interesse de programadores de sistemas básicos em linguagens como C.
- *Programação orientada a objetos*: é o assunto desse trabalho
- *RAD* (Rapid Application Development): significa desenvolvimento rápido de aplicações; ambientes que implementam RAD permitem o desenvolvimento de programas de entrada e saída de dados utilizando pouco ou nenhum código (o código é gerado automaticamente). Utiliza a programação visual.
- *Prototipagem*: um protótipo é uma aplicação que ainda não tem toda a funcionalidade final. Seu objetivo maior é o diálogo com o usuário e os refinamentos sucessivos. Tipicamente, sistemas são construídos desenhando suas telas e relatórios facilmente, com RAD, e vai evoluindo gradativamente à medida que seu desenvolvimento prossegue, até adquirir toda a funcionalidade desejada.
- *Arquitetura Cliente/Servidor*: os ambientes modernos permitem a divisão do programa entre código cliente e servidor, que pode ser executado em máquinas distintas em um ambiente distribuído. Os ambientes possuem drivers para acessar SQL de vários bancos de dados e conexão ODBC.
- *Sistemas abertos*: em geral o objetivo de aderir a padrões é alcançado nesses ambientes
- *Gerenciamento de equipes de desenvolvedores*: equipes de desenvolvedores trabalhando em um sistema complexo precisam de uma organização à nível de programas. Os ambientes modernos proporcionam facilidades de check-in/check-out (bloqueio para leitura e escrita de componentes do sistema), controle de versões e até o armazenamento dos sistemas integralmente em bancos de dados.

1.5. Programação Orientada a Objetos

Programação orientada a objetos (POO) é uma metodologia de programação adequada ao desenvolvimento de sistemas de grande porte, provendo modularidade e reusabilidade. A POO introduz uma abordagem na qual o programador visualiza seu programa em execução como uma coleção de objetos cooperantes que se comunicam através de mensagens. Cada um dos objetos é instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamento de herança. Existem alguns aspectos importantes na definição de POO:

- Usa **objetos**, e não funções ou procedimentos como seu bloco lógico fundamental de construção de programas
- Objetos comunicam-se através de **mensagens**
- Cada objeto é instância de uma **classe**
- Classes estão relacionadas com as outras via mecanismos de **herança**

POO é uma maneira de programar que difere substancialmente da programação tradicional, como em COBOL, Pascal, FORTRAN, C e outras linguagens. O que torna a POO diferente é que em vez de iniciar um programa escrevendo um laço principal e então todas as subrotinas que constituem a aplicação, inicia-se decidindo quais objetos a aplicação requer e então os objetos são caracterizados com propriedades e comportamento. Objetos são agrupados em uma hierarquia de classes, como a hierarquia existente no reino animal.

A expressão programação orientada a objetos é derivada do conceito de objeto utilizado na linguagem Simula 67. Nesta linguagem, a execução de um programa é realizada de maneira cooperativa, mas não paralela, seguindo uma organização em corrotinas, de uma coleção de objetos. Os objetos compartilhando uma estrutura comum são ditos instâncias de uma mesma classe.

Embora outras linguagens de programação tenham mostrado algumas tendências à orientação a objetos, Smalltalk é ainda a linguagem mais representativa do paradigma de objetos. Mais do que uma linguagem de programação, Smalltalk é um completo ambiente de programação refletindo a filosofia de orientação a objetos.

Programação orientada a objetos dá ênfase à estrutura de dados, adicionando funcionalidade ou capacidade de processamento a estas estruturas. Em linguagens tradicionais, a importância maior é atribuída a processos, e sua implementação em subprogramas. Em uma linguagem como Pascal, procedimentos ativos agem sobre dados passivos que foram passados a eles. Em linguagens orientadas a objetos, ao invés de passar dados a procedimentos, requisita-se que objetos realizem operações neles próprios.

Alguns aspectos são fundamentais na definição de programação orientada a objetos, que serão devidamente investigados neste trabalho:

- Abstração de dados
- Objetos
- Mensagens

- Classes
- Herança
- Polimorfismo

2. Conceitos básicos e terminologia

Para que se possa utilizar uma linguagem de programação orientada a objetos é necessário primeiramente conhecer os conceitos e a terminologia utilizada por este paradigma.

2.1. Abstração de dados

Um conceito que teve muita influência na POO chama-se programação com tipos abstratos de dados. Tipos abstratos de dados possuem características como *encapsulamento* e *modularidade*.

A noção de tipos de dados ocorre na maioria das linguagens de programação tradicionais. Na declaração do tipo de uma variável, delimita-se o conjunto de valores que ela pode assumir e as operações que ela pode sofrer. A especificação de um tipo de dados deve definir os objetos constituintes do tipo e as operações aplicáveis a estes objetos. Além disso, é possível estabelecer uma maneira de representação para os objetos.

Geralmente, uma linguagem de programação provê alguns tipos básicos pré-definidos (tipos primitivos) e ainda oferece mecanismos para definição de novos tipos de dados renomeando tipos existentes ou agregando alguns tipos primitivos e/ou definidos pelo usuário (em Pascal, por exemplo, utiliza-se o comando *type* para criar novos tipos de dados). Existem linguagens que além de proporcionarem tipos de dados primitivos e tipos de dados definidos pelo usuário, incorporam o conceito de tipos abstratos de dados. Tipos abstratos de dados envolvem a disponibilidade de dados e operações (comportamento) sobre estes dados em uma única unidade.

A abstração de dados é utilizada para introduzir um novo tipo de objeto, que é considerado útil no domínio do problema a ser resolvido. Os usuários do tipo abstrato de dados preocupam-se apenas com o comportamento dos objetos do tipo, demonstrado em termos de operações significativas para tais objetos, não necessitando conhecer como estes objetos estão representados ou como as operações são realizadas neles (ocultamento de informação). Portanto, uma abstração de dados consiste de um conjunto de valores e de operações que completamente caracterizam o comportamento dos objetos. Esta propriedade é garantida fazendo-se com que as operações sejam a única maneira de criar e manipular os objetos. Como conseqüência, é necessário incluir operações suficientes para proporcionar todas as possíveis ações que os objetos possam sofrer.

Mais precisamente, um tipo abstrato de dados pode ser definido como um tipo de dados que satisfaz as seguintes condições:

- A representação e definição do tipo e as operações nos objetos do tipo são descritos em uma única unidade sintática (como em uma *Unit*, em Turbo Pascal)
- A representação (implementação) dos objetos do tipo é escondida das unidades de programa que utilizam o tipo; portanto, as únicas operações possíveis em tais objetos são aquelas que fazem parte da definição (interface) do tipo.

O estilo de programação com tipos abstratos de dados inclui o princípio de **encapsulamento**, que proporciona ocultamento e proteção de informação, viabilizando a manutenção e facilitando a evolução de sistemas. Com o encapsulamento da estrutura de dados que representa os objetos e dos procedimentos que representam as possíveis operações sobre os objetos, tem-se que uma alteração na estrutura de dados provavelmente exigirá modificações nos procedimentos, porém o efeito dessas modificações fica restrito às fronteiras da unidade sintática que descreve o tipo (a *Unit*, no caso). Se a interface permanecer a mesma, as unidades de programa que utilizam o tipo não necessitam sofrer alterações.

Uma outra vantagem que pode ser citada é o considerável aumento de confiabilidade obtido através do princípio da proteção. Unidades de programa que utilizam um tipo não estão aptas a fazer modificações diretamente. Elas somente podem chamar as operações que estão disponíveis na interface, aumentando a integridade dos objetos.

2.2. Objetos

Na visão de uma linguagem imperativa tradicional (Pascal, C, COBOL, etc.), os *objetos* aparecem como uma única entidade autônoma que combina a representação da informação (estruturas de dados) e sua manipulação (procedimentos), uma vez que possuem capacidade de processamento e armazenam um estado local. Pode-se dizer que um objeto é composto de (Figura 2.1):

- *Propriedades*: são as informações, estruturas de dados que representam o estado interno do objeto. Em geral, não são acessíveis aos demais objetos.
- *Comportamento*: conjunto de operações, chamados de *métodos*, que agem sobre as propriedades. Os métodos são ativados (disparados) quando o objeto recebe uma *mensagem* solicitando sua execução. Embora não seja obrigatório, em geral uma mensagem recebe o mesmo nome do método que ela dispara. O conjunto de mensagens que um objeto está apto a receber está definido na sua *interface*.
- *Identidade*: é uma propriedade que diferencia um objeto de outro; ou seja, seu nome.

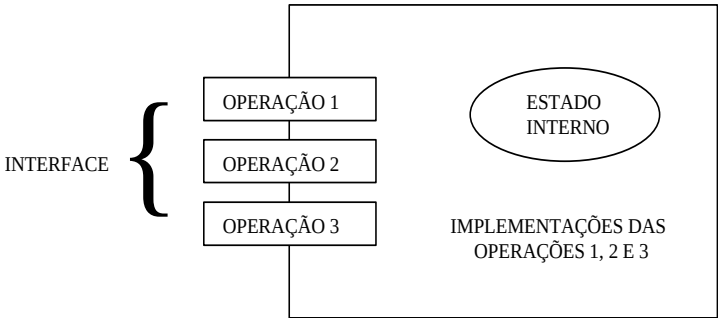


Figura 2.1. Estrutura de um objeto

Enquanto que os conceitos de dados e procedimentos são freqüentemente tratados separadamente nas linguagens de programação tradicionais, em POO eles são reunidos em uma única entidade: o objeto. A figura 2.2 apresenta outra visualização para um objeto.

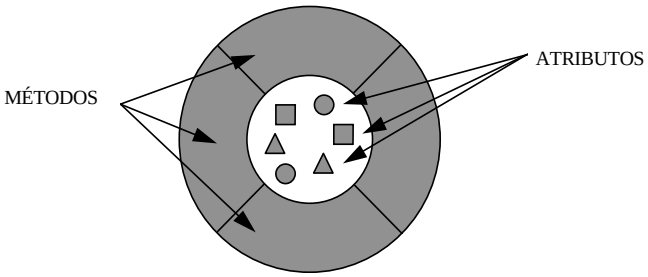


Figura 2.2. Outra representação para um objeto

No mundo real não é difícil a identificação de objetos (em termos de sistemas, objetos são todas as entidades que podem ser modeladas, não apenas os nossos conhecidos objetos inanimados). Como exemplo, em uma empresa pode-se identificar claramente objetos da classe (seção 2.4) *empregado*.

Um empregado possui uma identidade própria, seu nome: José da Silva. Possui também propriedades como: endereço, idade, dependentes, salário, cargo, entre outras. O comportamento pode ser determinado por operações como: aumentar salário, listar dependentes e alterar cargo. As propriedades somente podem ser manipuladas através das operações definidas na interface do objeto, de modo que a forma de armazenamento das propriedades e a implementação das operações são desconhecidas pelas outras entidades externas (encapsulamento de informações).

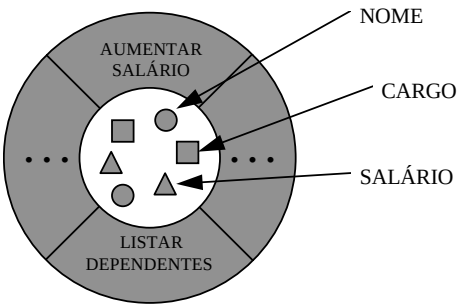


Figura 2.3. O objeto "Empregado"

Benefícios proporcionados pelos objetos:

Uma vez que objetos utilizam o princípio da abstração de dados (seção 2.1), o encapsulamento de informação proporciona dois benefícios principais para o desenvolvimento de sistemas:

- *Modularidade*: o código fonte para um objeto pode ser escrito e mantido independentemente da código fonte de outros objetos. Além disso, um objeto pode ser facilmente migrado para outros sistemas.
- *Ocultamento de informação*: um objeto tem uma interface pública que os outros objetos podem utilizar para estabelecer comunicação com ele. Mas, o objeto mantém informações e métodos privados que podem ser alterados a qualquer hora sem afetar os outros objetos que dependem dele. Ou seja, não é necessário saber como o objeto é implementado para poder utilizá-lo.

2.3. Mensagens

Um objeto sozinho não é muito útil e geralmente ele aparece como um componente de um grande programa que contem muitos outros objetos. Através da interação destes objetos pode-se obter uma grande funcionalidade e comportamentos mais complexos.

Objetos de software interagem e comunicam-se com os outros através de mensagens. Quando o objeto A deseja que o objeto B execute um de seus métodos, o objeto A envia uma mensagem ao objeto B (Figura 2.4). Algumas vezes o objeto receptor precisa de mais informação para que ele saiba exatamente o que deve fazer; esta informação é transmitida juntamente com a mensagem através de *parâmetros*.

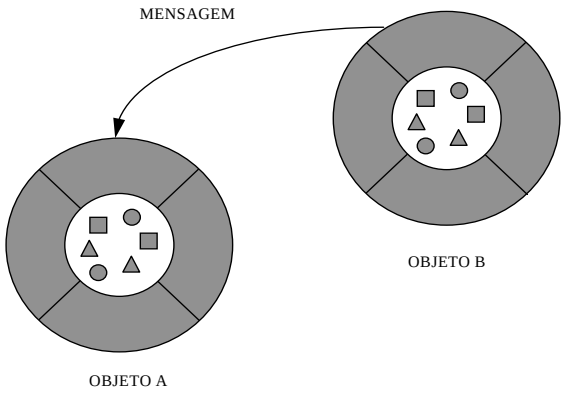


Figura 2.4. O envio de mensagens entre objetos

Uma mensagem é formada por três componentes básicos:

- o objeto a quem a mensagem é endereçada (receptor)
- o nome do método que se deseja executar
- os parâmetros (se existirem) necessários ao método

Estes três componentes são informação suficiente para o objeto receptor executar a método desejado. Nada mais é necessário. Então, objetos em processos

distintos ou ainda em máquinas distintas podem comunicar-se através do uso de mensagens.

Como exemplo, pode-se imaginar a realização de uma operação aritmética utilizando-se objetos, métodos e mensagens. Uma adição é realizada enviando-se uma mensagem a um objeto representando o número. A mensagem especifica que a operação desejada é a adição e também o número que deve ser somado ao objeto receptor. Então, a operação "*x+ y*" é interpretada como a mensagem "+" sendo enviada ao objeto "x" com o parâmetro "y". Pode parecer estranho à primeira vista, mas em SmallTalk operações aritméticas são realmente tratadas dessa forma.

A título de comparação com as linguagens convencionais, uma mensagem se comporta como uma chamada de subrotina (procedimento ou função). A sintaxe em geral também é muito parecida.

Benefícios proporcionados pelas mensagens:

- Desde que tudo que um objeto pode fazer é expressado pelos seus métodos, a troca de mensagens suporta todos os tipos possíveis de interação entre objetos
- Para enviar e receber mensagens, objetos não precisam estar no mesmo processo ou máquina (ex: CORBA)

2.4. Métodos

Um método implementa algum aspecto do comportamento do objeto. Comportamento é a forma como um objeto age e reage, em termos das suas trocas de estado e troca de mensagens.

Um método é uma função ou procedimento que é definido na classe e tipicamente pode acessar o estado interno de um objeto da classe para realizar alguma operação. Pode ser pensado como sendo um procedimento cujo primeiro parâmetro é o objeto no qual deve trabalhar. Este objeto é chamado receptor (seção 2.3). Abaixo é apresentada uma notação possível para o envio de uma mensagem (invocação do método)

receptor.nome_da_mensagem(par1, par2, par3)

Muitas vezes quando se trabalha com objetos deseja-se realizar certas operações no momento da criação e destruição de um objeto. Em algumas linguagens é dado um tratamento de modo a ficar transparente ao programador a sua implementação. Em outras, é necessário utilizar métodos especiais para este fim, chamados de **construtores** e **destrutores**.

Construtores são usados para criar e inicializar objetos novos. Tipicamente, a inicialização é baseada em valores passados como parâmetros para o construtor. Destrutores são usados para destruir objetos. Quando um destrutor é invocado, as ações definidas pelo usuário são executadas, e então a área de memória alocada para o objeto é liberada. Em algumas linguagens, como C++, o construtor é chamado automaticamente quando um objeto é declarado. Em outras, como Object Pascal, é necessário chamar explicitamente o construtor antes de poder utilizá-lo.

Um exemplo de utilização de construtores e destrutores seria gerenciar a quantidade de objetos de uma determinada classe que já foram criados até o momento. No construtor pode-se colocar código para incrementar uma variável e no destrutor o código para decrementá-la. Na seção 4 são apresentados exemplos da utilização de construtores e destrutores.

2.5. Classes

Objetos de estrutura e comportamento idênticos são descritos como pertencendo a uma classe, de tal forma que a descrição de suas propriedades pode ser feita de uma só vez, de forma concisa, independente do número de objetos idênticos em termos de estrutura e comportamento que possam existir em uma aplicação. A noção de um objeto é equivalente ao conceito de uma variável em programação convencional, pois especifica uma área de armazenamento, enquanto que a classe é vista como um tipo abstrato de dados, uma vez que representa a definição de um tipo.

Cada objeto criado a partir de uma classe é denominado de *instância* dessa classe. Uma classe provê toda a informação necessária para construir e utilizar objetos de um tipo particular, ou seja, descreve a forma da memória privada e como se realizam as operações das suas instâncias. Os métodos residem nas classes, uma vez que todas as instâncias de uma classe possuem o mesmo conjunto de métodos, ou seja, a mesma interface.

Cada instância pertence a uma classe e uma classe pode possuir múltiplas instâncias. Devido ao fato de todas as instâncias de uma classe compartilharem as mesmas operações, qualquer diferença de respostas a mensagens aceitas por elas, é determinada pelos valores das variáveis de instância.

Em algumas linguagens, aparece também a definição de variáveis de classe. Variáveis de classe surgem quando se tem a visão de classes sendo manipuladas como objetos. Esta abordagem se torna útil ao se tentar manter classes contendo informação. Como exemplo, uma classe poderia armazenar o número de objetos que tenham sido instanciados da classe até um certo momento. Cita-se, ainda, a importância de se ter classes como objetos para modelar a entidade que recebe uma mensagem requisitando a criação de um novo objeto (como em Object Pascal).

A Figura 2.5 ilustra o relacionamento entre classes e objetos. Cada objeto instanciado a partir de uma classe possui as propriedades e comportamento definidos na classe, da mesma maneira que uma variável incorpora as características do seu tipo. A existência de classes proporciona um ganho em reusabilidade. pois o código das operações e a especificação da estrutura de um número potencialmente infinito de objetos estão definidos em um único local, a classe. Cada vez que um novo objeto é instanciado ou que uma mensagem é enviada, a definição da classe é reutilizada. Caso não existissem classes, para cada novo objeto criado, seria preciso uma definição completa do objeto.

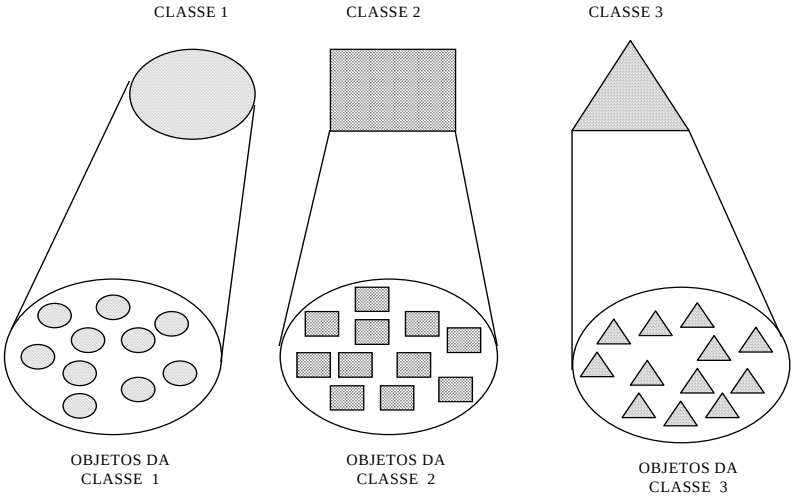


Figura 2.5. Relacionamento entre classes e objetos

No exemplo de uma empresa seria possível criar classes para várias entidades a ser modeladas, como: empregados, departamentos, filiais, produtos, documentos fiscais, etc. Na definição da classe *empregado*, são especificados as suas propriedades e comportamento e escrito o código fonte da sua implementação. Quando um novo empregado é admitido na empresa, ele é criado como uma instância da classe *empregado*, e automaticamente possui todas as características da classe.

Metaclasses

Uma metaclasse é uma classe de classes. Pode-se julgar conveniente que, em uma linguagem ou ambiente, classes também possam ser manipuladas como objetos. Por exemplo, uma classe pode conter variáveis contendo informações úteis, como:

- o número de objetos que tenham sido instanciados da classe até certo instante;
- um valor médio de determinada propriedade, calculado sobre os valores específicos desta propriedade nas instâncias (por exemplo, média de idade de empregados).

Essas variáveis não devem ser criadas para cada objeto (instância), mas para cada classe deve existir apenas uma cópia de cada. Para implementar isto é conveniente considerar uma classe como sendo também um objeto. Neste caso, a classe precisará ser instância de outra classe. Esta classe de classes é chama de metaclasse.

Benefícios proporcionados pelas classes:

O maior benefício proporcionado pela utilização das classes é a reusabilidade de código, uma vez que todos os objetos instanciados a partir dela incorporam as suas propriedades e seu comportamento.

2.6. Herança

O mecanismo de herança permite a reutilização das propriedades de uma classe na definição de outra. A classe mais generalizada é chamada *superclasse* e a

mais especializada, *subclasse*. No desenvolvimento de software, herança surge como a maneira para garantir que caixas pretas construídas não se tornem caixas isoladas. Através da herança, novas caixas pretas são obtidas aproveitando-se operações já implementadas, proporcionando a *reusabilidade de código* existente. Assim como no mundo dos seres humanos, o objeto descendente não tem nenhum trabalho para receber a herança. Ele a recebe simplesmente porque é um objeto descendente.

Herança é a contribuição original do paradigma de objetos, que o diferencia da programação com tipos abstratos de dados. Ela viabiliza a construção de sistemas a partir de componentes reusáveis. A herança não só suporta a reutilização entre sistemas, mas facilita diretamente a extensabilidade em um mesmo sistema. Diminui a quantidade de código para adicionar características a sistemas já existentes e também facilita a manutenção de sistemas, tanto provendo maior legibilidade do código existente, quanto diminuindo a quantidade de código a ser acrescentada.

A herança não é limitada a apenas um nível; a árvore de herança, ou *hierarquia de classes* pode ser tão profunda quanto for necessário. Métodos e variáveis internas são herdados por todos os objetos dos níveis para baixo. Quanto mais para baixo na hierarquia uma classe estiver, mais especializado o seu comportamento. Várias subclasses descendentes podem herdar as características de uma superclasse ancestral, assim como vários seres humanos herdam as características genéticas de um antepassado.

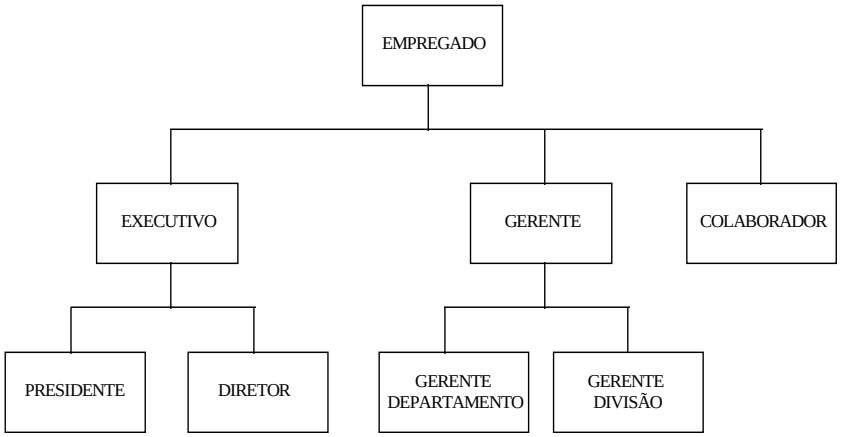


Figura 2.6. Herança na classe “Empregado”

Na figura 2.6 pode-se ver a hierarquia de classes para a classe “Empregado”. As classes Executivo, Gerente e Colaborador herdam todos os métodos e variáveis da classe Empregado; além disso, podem redefini-los e criar novas características. A classe Presidente, também herda todas as características da classe Executivo, que por sua vez já herdou da classe Empregado.

A herança é interpretada da seguinte maneira: se B é subclasse de A (conseqüentemente, A é superclasse de B), então:

- os objetos da classe B suportam todas as operações suportadas pelos objetos da classe A, exceto aquelas redefinidas
- as variáveis de instância de B incluem todas as variáveis de instância de A.

Analisando os itens acima, tem-se que uma subclasse compartilha o comportamento e o estado de sua superclasse, podendo redefini-lo para seu próprio

uso. Além disso, é possível acrescentar novas características que identificam um comportamento próprio. A maior parte das linguagens orientadas a objetos utiliza herança como uma técnica para prover suporte à *especialização*, e não permitem que uma classe exclua uma operação herdada.

A utilização de herança em linguagens orientadas a objetos pode afetar o encapsulamento de informação. Além dos clientes da classe que simplesmente requisitam operações sobre instâncias da classe, surge uma nova categoria de clientes que são as classes definidas através da herança, ou seja, as subclasses. Geralmente, às subclasses é permitido o total acesso à representação das variáveis de instâncias definidas na sua superclasse. Desta forma, compromete-se o princípio de encapsulamento, uma vez que as variáveis de instância de uma classe serão parte implícita da interface que ela proporciona para as suas possíveis subclasses. Mudanças na implementação de uma classe, como por exemplo a remoção de uma variável de instância, poderá comprometer a implementação de suas subclasses. Assim, não existe mais a liberdade para modificar a representação de uma classe em provocar efeitos nos seus clientes.

Herança não necessariamente está limitada a uma única superclasse. Uma subclasse pode herdar características de diversas superclasses, introduzindo o conceito de *herança múltipla*. Indiscutivelmente, é obtido um maior potencial para o compartilhamento de código, porém a possibilidade de conflitos, como o confronto de nomes entre as operações herdadas, entre múltiplas superclasses aumenta a complexidade dos sistemas.

Benefícios proporcionados pela herança:

- Subclasses provêem comportamentos especializados tomando-se como base os elementos comuns definidos pela superclasse. Através do uso da herança, pode-se reutilizar o código da superclasse várias vezes.
- Pode-se implementar classes que definem comportamentos “genéricos” (chamadas de *classes abstratas*). A essência da superclasse é definida e pode ser parcialmente implementada, mas a maior parte da classe é deixada indefinida e não implementada. Os detalhes são definidos em subclasses especializadas, que podem ser implementadas inclusive por outros programadores de acordo com sua necessidade.

2.6. Polimorfismo

Polimorfismo refere-se à capacidade de dois ou mais objetos responderem à mesma mensagem, cada um a seu próprio modo. A utilização da herança torna-se fácil com o polimorfismo. Desde que não é necessário escrever um método com nome diferente para responder a cada mensagem, o código é mais fácil de entender. Por exemplo, sem polimorfismo, para inserir um novo empregado, seria necessário o seguinte código:

*Colaborador1.**InsereColaborador***
*Gerente1.**InsereGerente***
*Presidente1.**InserePresidente***

Neste caso, Colaborador1, Gerente1 e Presidente1 são objetos das respectivamente das classes Colaborador, Gerente e Presidente. Com o polimorfismo, não é necessário ter um método diferente para cada tipo de objeto. Pode-se simplesmente escrever o seguinte:

*Colaborador.**Insere***
*Gerente.**Insere***
*Presidente.**Insere***

Neste exemplo, os três diferente empregados têm três diferentes métodos para ser inseridos, embora o tenha sido utilizado o método com o mesmo nome. No código para esses empregados, seria necessário escrever três métodos diferentes: um para cada tipo de empregado. O mais importante a ser observado é que as três rotinas compartilham o mesmo nome.

O polimorfismo obviamente não nos auxilia a reutilizar o código. Em vez disso, ele é útil na compreensão de programas. Desde que uma grande quantidade de objetos compartilham o mesmo comportamento com implementações ligeiramente diferentes, encontrar uma maneira de diminuir o número de linhas de código para invocar estes comportamentos auxilia imensamente. Os programadores que são usuários da classe Empregado não precisam saber que a maneira de inserir um empregado é diferente; quando eles querem inserir, somente utilizam o método *Insere*.

Se considerarmos que o método *Insere* está sendo utilizado dentro de uma estrutura de repetição, fica bastante aparente o ganho obtido em legibilidade e tamanho do código. Sem a utilização de polimorfismo esta estrutura seria:

```
Para todos os Empregados Faça
        Se Empregado é Presidente Então
                Empregado[i].InserePresidente
        Se Empregado é Gerente Então
                Empregado[i].InsereGerente
        Se Empregado é Colaborador Então
                Empregado[i].InsereColaborador
        Fim Se
Fim Para
```

Com a utilização do polimorfismo seria possível utilizar o seguinte código:

```
Para todos os Empregados Faça
        Empregado[i].Insere
Fim Para
```

O polimorfismo visto nos exemplos acima implica na existência de métodos com implementação diferente com o mesmo nome em classes distintas. Outra forma simples de polimorfismo permite a existência de vários métodos com o mesmo nome, definidos na mesma classe, que se diferenciam pelo tipo ou número de parâmetros suportados. Isto é conhecido como *polimorfismo paramétrico*, ou *sobrecarga de operadores* (“overloading”). Neste caso, uma mensagem poderia ser enviada a um objeto com parâmetros de tipos diferentes (uma vez inteiro, outra real, por exemplo), ou com número variável de parâmetros. O nome da mensagem seria o mesmo, porém o método invocado seria escolhido de acordo com os parâmetros enviados.

Dependendo das características da linguagem e do tipo de polimorfismo suportado por ela, a implementação do polimorfismo pode exigir facilidades proporcionadas pela **ligação dinâmica**. De um modo geral, uma ligação (*binding*) é uma associação, possivelmente entre um atributo e uma entidade ou entre uma operação e um símbolo. Uma ligação é dita estática se ocorre em tempo de compilação ou de interligação (*linking*) e não é mais modificada durante toda a execução do programa. A ligação dinâmica é feita em tempo de execução ou pode ser alterada no decorrer da execução do programa.

Com a ligação dinâmica, a ligação do operador a uma particular operação pode ser feita em tempo de execução, isto é, o código compilado para um tipo abstrato de dados pode ser usado para diferentes tipos de dados, aumentando consideravelmente sua reusabilidade. Entretanto, o uso de ligação dinâmica compromete a eficiência de implementação.

Benefícios proporcionados pelo polimorfismo

- Legibilidade do código: a utilização do mesmo nome de método para vários objetos torna o código de mais fácil leitura e assimilação, facilitando muito a expansão e manutenção dos sistemas.
- Código de menor tamanho: o código mais claro torna-se também mais enxuto e elegante. Pode-se resolver os mesmos problemas da programação convencional com um código de tamanho reduzido.

2.7. Relações entre Objeto, Classe e Herança

Objetos, classes e o mecanismo de herança permitem a definição de hierarquias de abstrações, que facilitam o entendimento e o gerenciamento da complexidade dos sistemas estudados. Isto porque classes agrupam objetos com características iguais, enquanto herança estrutura classes semelhantes.

A capacidade em classificar objetos e classes concede grande poder de modelagem conceitual e classificação, podendo expressar relações entre comportamentos, tais como classificação/instanciação, generalização/especialização e agregação/composição. Facilita a compreensão humana do domínio estudado e também o desenvolvimento de programas que o satisfaça. Pode-se dizer que a grande vantagem do paradigma de objetos é a possibilidade de expressar diretamente este poder de modelagem conceitual numa linguagem de programação orientada a objetos. Assim, o modelo conceitual proposto durante a etapa de análise não se perde nas etapas de projeto e implementação. O que em geral ocorre é a sua extensão.

2.7.1. Classificação/Instanciação

A capacidade de classificar objetos (em classes) permite expressar relações do tipo classificação/instanciação. O relacionamento é feito a partir da observação de diversos fenômenos para categorização dos mesmos em grupos (classes), com base no conjunto de propriedades comuns a todos. Por exemplo, dois computadores, *IBM PC* e *Machintosh*, podem ser classificados como **instâncias** (objetos, modelos, ou espécimes) da classe (categoria) Microcomputador (Figura 2.7). A relação inversa é

a de instanciação de uma publicação (*IBM PC*, por exemplo) a partir da classe Microcomputador.

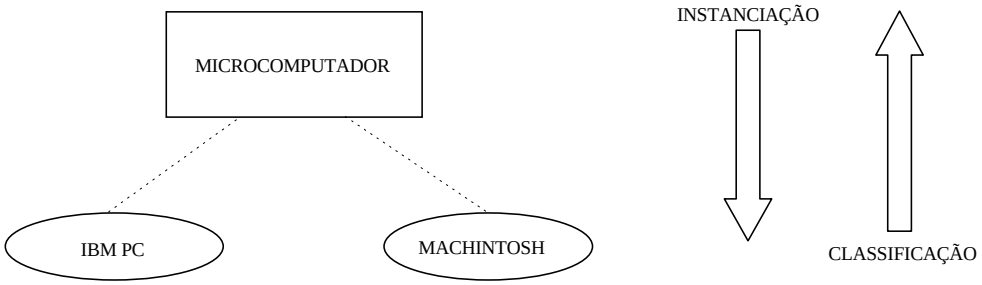


Figura 2.7. Relação de classificação/instanciação

2.7.2. Generalização/Especialização

Este tipo de relação ocorre quando, a partir a observação de duas classes, abstraímos delas uma classe mais genérica. Por exemplo, as classes Microcomputador e Mainframe podem ser considerados casos especiais da classe Computador. Esta classe é considerada uma **generalização** das duas primeiras, que são chamadas de **especializações** da classe Computador (Figura 2.8). A idéia da generalização/especialização é a base para a classificação das espécies nas ciências naturais. Do ponto de vista de propriedades, o pressuposto é que as subclasses tenham todas as propriedades das classes de quem elas são especializações. Deve haver pelo menos uma propriedade que diferencie duas classes especializadas (subclasses) a partir da mesma classe genérica (superclasse). Este é o tipo de relação utilizado com o mecanismo de herança.

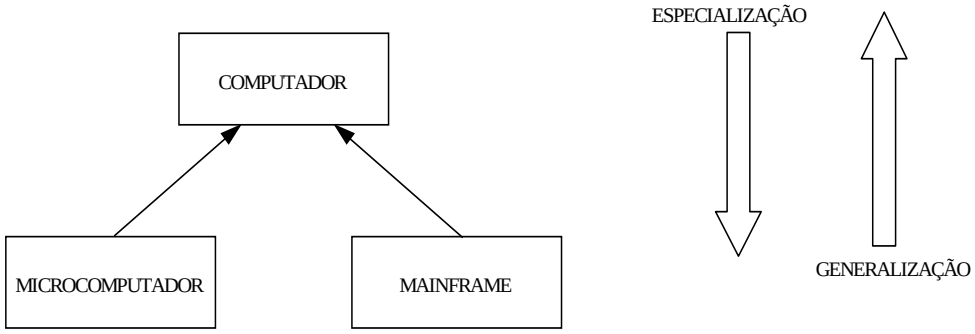


Figura 2.8. Relação de generalização/especialização

2.7.3. Composição/Decomposição

A relação de composição (ou agregação)/decomposição permite que objetos sejam compostos pela agregação de outros objetos ou componentes. Neste relacionamento, são determinados que instâncias (objetos) de uma classe são compostas por instâncias de outras classes. Essa operação é denominada **decomposição** e a relação inversa, a formação de uma nova classe como um agregado de classes preexistentes, é denominada **composição**. Por exemplo, instâncias da

classe Microcomputador são compostas por, entre outras, instâncias das classes Teclado e Vídeo (Figura 2.9).

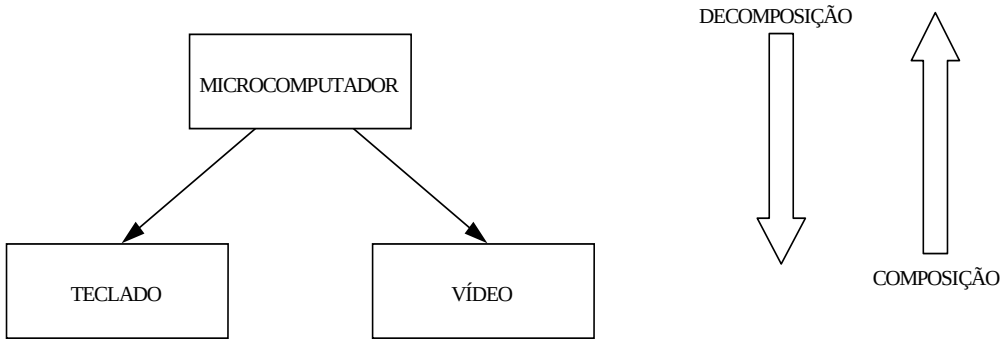


Figura 2.9. Relação de composição/decomposição

3. Avaliação da POO

O paradigma de objetos não é apenas um nome pomposo. Seu objetivo claro é melhorar a produtividade dos programadores no desenvolvimento a sistemas. A POO realmente torna mais rápidas as atividades de programação e manutenção de sistemas de informação. Assim como a programação estrutura melhorou a qualidade dos sistemas em relação aos métodos praticados anteriormente (programas estilo "spagueti" recheados de comandos *goto*), a POO surge como um modelo promissor para solucionar as necessidades crescentes da nossa sociedade por sistemas informatizados.

A grande vantagem do paradigma de objetos é o seu caráter **unificador**: trata todas as etapas do desenvolvimento de sistemas e ambientes sob uma única abordagem. Nesse sentido, podemos ter análise, projeto, programação, banco de dados e ambientes orientados a objetos, eliminando as diferenças de "impedância" entre eles.

3.1. Vantagens da POO

A POO tem alcançado tanta popularidade devido às vantagens que ela traz. Entre elas podemos citar:

- Reusabilidade de código
- Escalabilidade de aplicações
- Mantenabilidade
- Apropriação

A reusabilidade de código é, sem dúvida, reconhecida como a maior vantagem da utilização de POO, pois permite que programas sejam escritos mais rapidamente. Todas as empresas sofrem de deficiência em seus sistemas informatizados para obter maior agilidade e prestar melhores serviços a seus clientes. Um levantamento feito na AT&T, a gigante das telecomunicações nos EUA, identificou uma deficiência da ordem de bilhões de linhas de código. Uma vez que a demanda está sempre

aumentando, procura-se maneiras de desenvolver sistemas mais rapidamente, o que está gerando uma série de novas metodologias e técnicas de construção de sistemas (por exemplo, ferramentas CASE). A POO, através da reusabilidade de código, traz uma contribuição imensa nesta área, possibilitando o desenvolvimento de novos sistemas utilizando-se muito código já existente. A maior contribuição para reusabilidade de código é apresentada pela herança.

Escalabilidade pode ser vista como a capacidade de uma aplicação crescer facilmente sem aumentar demasiadamente a sua complexidade ou comprometer o seu desempenho. A POO é adequada ao desenvolvimento de grandes sistemas uma vez que pode-se construir e ampliar um sistema agrupando objetos e fazendo-os trocar mensagens entre si. Esta visão de sistema é uniforme, seja para pequenos ou grandes sistemas (logicamente, deve-se guardar as devidas proporções).

O encapsulamento proporciona ocultamento e proteção da informação. Acessos a objetos somente podem ser realizados através das mensagens que ele está habilitado a receber. Nenhum objeto pode manipular diretamente o estado interno de outro objeto. De modo que, se houver necessidade de alterar as propriedades de um objeto ou a implementação de algum método, os outros objetos não sofrerão nenhum impacto, desde que a interface permaneça idêntica. Isto diminui em grande parte os esforços despendidos em manutenção. Além disso, para utilizar um objeto, o programador não necessita conhecer a fundo a sua implementação.

O polimorfismo torna o programa mais enxuto, claro e fácil de compreender. Sem polimorfismo, seriam necessárias listas enormes de métodos com nomes diferentes mas comportamento similar. Na programação, a escolha de um entre os vários métodos seria realizada por estruturas de múltipla escolha (*case*) muito grandes. Em termos de manutenção, isto significa que o programa será mais facilmente entendido e alterado.

A herança também torna a manutenção mais fácil. Se uma aplicação precisa de alguma funcionalidade adicional, não é necessário alterar o código atual. Simplesmente cria-se uma nova geração de uma classe, herdando o comportamento antigo e adiciona-se novo comportamento ou redefine-se o comportamento antigo.

Outra vantagem da POO é também uma desvantagem: apropriação. Problemas do mundo real são freqüentemente solucionados facilmente utilizando as técnicas de orientação a objetos. As nossas mentes são classificadores naturais de coisas. Tipicamente nós classificamos coisas em hierarquias de classes. A POO leva vantagem desse fato e nos fornece uma estrutura que funciona como a nossa mente. Apropriação significa agrupar os objetos de comportamento similar em classes e essas classes em hierarquias de herança.

3.2. Desvantagens da POO

Apesar de das inúmeras vantagens, a POO tem também algumas desvantagens, que incluem:

- Apropriação
- Fragilidade
- Linearidade de desenvolvimento

A apropriação é apresentada tanto como uma vantagem como uma desvantagem, porque a POO nem sempre soluciona os problemas elegantemente. Enquanto que a mente humana parece classificar objetos em categorias (classes) e agrupar essas classes em relacionamentos de herança, o que ela realmente faz não é tão simples. Em vez disso, objetos com características mais ou menos similares, e não precisamente definidas, são reunidos em uma classificação. A POO requer definições precisas de classes; definições flexíveis e imprecisas não são suportadas. Na mente humana, essas classificações podem mudar com o tempo. Os critérios para classificar objetos podem mudar significativamente. A apropriação utilizada na POO torna-a muito rígida para trabalhar com situações dinâmicas e imprecisas.

Além disso, algumas vezes não é possível decompor problemas do mundo real em uma hierarquia de classes. Negócios e pessoas têm freqüentemente regras de operações sobre objetos que desafiam uma hierarquia limpa e uma decomposição orientada a objetos. O paradigma de objetos não trata bem de problemas que requerem limites nebulosos e regras dinâmicas para a classificação de objetos.

Isto leva ao próximo problema com POO: fragilidade. Desde que uma hierarquia orientada a objetos requer definições precisas, se os relacionamentos fundamentais entre as classes chave mudam, o projeto original orientada a objetos é perdido. Torna-se necessário reanalisar os relacionamentos entre os objetos principais e reprojetar uma nove hierarquia de classes. Se existir uma falha fundamental na hierarquia de classes, o problema não é facilmente consertado. A mente humana adapta-se continuamente, e geralmente adequadamente, a situações novas. Ela encontra maneiras de classificar objetos no ambiente automaticamente. Enquanto a nossa mente tem essa capacidade, os ambientes POO não são tão bem equipados.

Em virtude dessa fragilidade, a POO requer análise e projeto frontal para assegurar que a solução é adequada. Com isto existe uma tendência em criar uma abordagem linear de desenvolvimento, em vez de cíclica. Infelizmente, alguns problemas são “perversos”, o que significa que não se sabe como resolver um problema até efetivamente resolvê-lo. Tais problemas desafiam até mesmo os melhores projetistas e analistas de sistemas. Utilizar a abordagem de Desenvolvimento Rápido de Aplicações (RAD - Rapid Application Development, é utilizada pelo Delphi, SQL-Windows e outros ambientes de desenvolvimento para Windows adequados à prototipagem) com ciclos entre o projeto do protótipo, construção e realimentação do usuário é algumas vezes uma boa maneira de resolver problemas “perversos”. Entretanto, a POO necessita de um projeto cuidadoso da hierarquia de classes, o que pode elevar os custos da sua utilização em um ambiente RAD.

3.3. O estilo de programação com POO

O estilo de programação induzido pelo paradigma de objetos tem características muito precisas:

- *Modularidade*: o estilo impõe alta modularidade na tarefa de projeto e programação. O conceito de objetos e sua descrição em classes incorporando dados e operações constituem um critério de modularização altamente efetivo e propiciam uma boa tradução prática de encapsulamento de informações.

- *Suporte a generalização/especialização*: o estilo suporta diretamente a estruturação de conceitos em hierarquias de generalização/especialização que representa a deficiência fundamental de abordagens de programação limitadas a tipos abstratos de dados (TADs).
- *Visão balanceada entre dados e processos*: o estilo tenta induzir, ao contrário de outros métodos, uma visão integrada de dados e processos na modelagem conceitual de aplicações.
- *Composição "botton-up" de aplicação*: o estilo privilegia a composição de uma nova aplicação a partir de composição de classes de aplicações preexistentes. Nesse sentido, a composição da aplicação é "bottom-up" (de baixo para cima, ao contrário de "top-down").
- *A atividade incremental e evolutiva*: o estilo favorece uma abordagem incremental de programação, onde pequenas modificações podem ser rapidamente efetuadas e testadas, quer a partir da criação de novas classes, quer a partir da rápida especialização de um classe preexistente.
- *Reusabilidade de código*: o estilo favorece a reutilização de código já existente, na forma de bibliotecas de classes que serão diretamente instanciadas ou modificadas pelo usuário.
- *Robustez*: o estilo favorece a criação de aplicações robustas, ou seja, muito estáveis em situações sujeitas a falhas. A maioria das linguagens orientadas a objetos possui mecanismos de tratar adequadamente erros vindos do usuário e também exceções do sistema.
- *Programação em grande escala*: o estilo é adequado ao desenvolvimento de sistemas grandes e complexos com equipes formadas por vários programadores. Algumas características apresentadas acima contribuem substancialmente para este item, como a modularidade e a reusabilidade.

3.4. Programação Dirigida por Eventos

Existe uma confusão muito comum atualmente com relação à POO e Programação Dirigida por Eventos (PDE). Os dois conceitos costumam ser usados indisciplinadamente e também "misturados". Um erro comum é pensar que todas as linguagem de programação para Windows (que implementa a PDE) são também orientadas a objetos.

A linguagem SmallTalk, que é a linguagem orientada a objetos mais pura nasceu em conjunto com um ambiente gráfico de desenvolvimento. A seguir alguns sistemas copiaram a idéia das interfaces gráficas de usuário(GUI), alguns implementando POO, outros não. Na realidade as técnicas de orientação a objetos trazem muitos benefícios às interfaces de usuário, como será visto na seção 5.

Os conceitos de POO foram apresentados nas seções 2 e 3 e dizem respeito à objetos, classes, herança, polimorfismo, etc. A PDE diz respeito à maneira como a execução dos programas é tratada nos ambientes de desenvolvimento e contrasta diretamente com a programação seqüencial. POO e PDE representam aspectos

distintos porém complementares, que utilizados em conjunto obtêm alto grau de sinergia nos resultados apresentados.

Um determinado ambiente de desenvolvimento, como pode ser visto na Tabela 3.1, pode ter características de POO e PDE, de apenas uma delas, ou de nenhuma.

Ambiente	POO	PDE
COBOL tradicional	NÃO	NÃO
Visual Basic	NÃO	SIM
Turbo C++	SIM	NÃO
Delphi	SIM	SIM

Tabela 3.1. Ambientes POO e PDE

A seguir será dada uma pequena visão comparativa entre programação seqüencial e programação dirigida por eventos.

Programação Seqüencial

Programação Seqüencial é o que se imagina quando se pensa em programação. É o modelo básico para os programadores desde que a programação começou, e é fácil de conceituar. Em programação seqüencial, há um início, meio e fim. Ela é unidirecional. Tipicamente um programa seqüencial possui os seguintes passos:

1. Inicializar todas as variáveis
2. Ler os dados
3. Aplicar um algoritmo sobre os dados
4. Escrever os resultados

Onde o algoritmo poderia ser qualquer coisa, desde ordenar dados alfabeticamente até calcular horóscopo.

Em um nível fundamental, é exatamente isso o que ocorre dentro do computador. Pode-se ver um computador realizando muitas atividades ao mesmo tempo, mas uma CPU pode somente realizar efetivamente uma por vez. Parece que elas são executadas ao mesmo tempo porque o computador (o sistema operacional, na realidade) troca rapidamente de uma tarefa para outra

O modelo Seqüencial continua ainda sendo “verdadeiro”, mas algumas vezes é útil olhar de um ponto de vista diferente, ou de um nível de abstração diferente.

Programação Dirigida por Eventos

Programação Dirigida por Eventos é o modelo que tipicamente é utilizado pelos sistemas operacionais e programas aplicativos atuais. Na Programa Dirigida por Eventos, não existe um foco de processamento linear que o programa segue. Um programa é configurado como uma coleção de tratadores (*handlers*) que são utilizados quando os vários eventos ocorrem. Um tratador de eventos constitui-se basicamente em subrotina em um programa (um procedimento ou uma função). Um programa dirigido por eventos pode ser imaginado como tendo os seguintes passos:

1. Inicializar
2. Configurar a interface

3. Esperar por um evento
4. Quando um evento ocorrer, deve-se tratá-lo
5. Ir para o passo 3 (o estado de espera)

Por exemplo, um evento pode ser: o usuário “clica” em um ícone, o usuário escolhe “Novo” em um menu, ou o usuário pressiona a tecla “c”.

A principal diferença entre esses dois paradigmas é que em Programação Dirigida por Eventos, “qualquer coisa pode acontecer”. Um usuário (ou um programa, ou um computador através de um rede, etc.) pode causar um determinado evento dentre um vasto repertório de possibilidades. Os eventos disponíveis (que são tratados) são aqueles geralmente que o programador achar necessários.

Pode-se notar que a Programação Dirigida por Eventos constitui-se apenas em uma camada no topo da Programação Seqüencial. Um programa é constituído de muitos tratadores de eventos, que são programas seqüenciais, em vez de um único grande programa seqüencial. A Programação Dirigida por Eventos é então justamente um nível de abstração em cima do nosso modelo seqüencial básico, que torna mais fácil a tarefa de projetar e escrever programas. A linguagem de programação utilizada para implementar os tratadores de eventos pode seguir o paradigma procedimental tradicional (linguagens como Pascal, C, Cobol, Fortran, etc.), orientado a objetos além de outros possíveis.

4. Linguagens Típicas

Com relação ao suporte a objetos, pode-se classificar as linguagens de programação em: *não baseadas em objetos*, *baseadas em objetos*, *baseadas em classes* e *orientadas a objetos*.

Uma linguagem que não suporte objetos como uma construção primitiva é, obviamente, não baseada em objetos. Exemplos de linguagens não baseadas em objetos são Pascal, C, FORTRAN e COBOL.

Caso uma linguagem suporte objetos, então ela é denominada como sendo baseada em objetos. As linguagens baseadas em objetos incluem Modula-2, ADA, CLU, Simula, SmallTalk, C++ e Object Pascal.

Para ser considerada uma linguagem baseada em classes é necessário que todo objeto pertença a alguma classe. CLU, Simula, SmallTalk, C++ e Object Pascal são também linguagens baseadas em classes, mas não o são o ADA e Modula-2. Uma linguagem orientada a objetos é baseada em classes dispostas em hierarquias. SmallTalk, C++ e Object Pascal são orientadas a objetos, uma vez que suas classes suportam o mecanismo de herança. CLU é baseada em classes, mas não orientada a objetos, uma vez que suas classes não suportam herança.

Linguagens baseadas em classes são um subconjunto próprio das linguagens baseadas em objetos, enquanto linguagens orientadas a objetos são um subconjunto próprio das linguagens baseadas em classes. A Figura 4.1 mostra o relacionamento entre as classes de linguagens de programação baseadas a objetos.

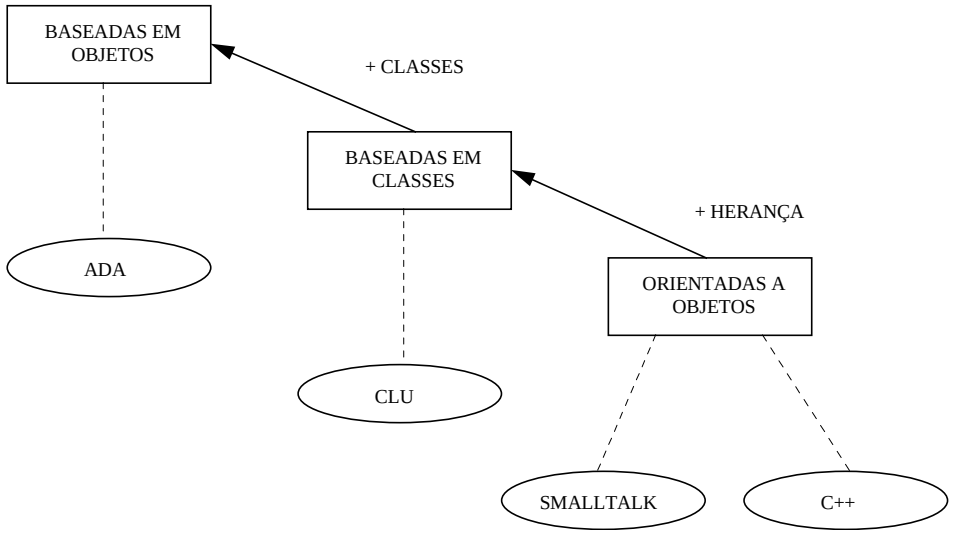


Figura 4.1. Relacionamento entre classes de linguagens baseadas em objetos

Estas classes de linguagens são úteis para distinguir as diferentes linguagens baseadas em objetos existentes, como base em suas características, relacionamento estas distinções com a capacidade da linguagem em suportar determinada metodologia. Por isso, deve-se considerar o efeito de objetos, classes e herança na metodologia de programação.

4.1. A linguagem de programação SmallTalk

A linguagem de programação Smalltalk é inteiramente implementada com base no modelo de objetos comunicando-se seqüencialmente através de mensagens. A idéia fundamental caracterizando a linguagem é a *uniformidade*. Um objeto representa um componente do sistema, isto é, representa números, filas, diretórios de discos, compiladores, etc.

Em SmallTalk o paradigma de objetos é levado às últimas conseqüências. Todas as entidades manipuladas por um programa são objetos ou referências a objetos. A construção 5 + 2, por exemplo, que em outras linguagens geralmente é considerada uma expressão aritmética, é interpretada em SmallTalk como um objeto representando o inteiro 5 ao qual é enviada a mensagem "+" cujo argumento é outro objeto representando o valor inteiro 2. Nesse caso, o objeto que representa o valor 5 efetua a operação de soma e retorna como resposta um objeto representando o valor inteiro 7.

As entidades básicas manipuladas em Smalltalk refletem os conceitos já apresentados. Um objeto consiste de uma área de memória (variáveis de instância) e de operações (métodos). Uma mensagem é vista como uma requisição para um objeto realizar uma de suas operações. A definição da interface de um objeto é feita através das mensagens que o objeto pode responder. Esta interface restringe a manipulação da área privada de um objeto às operações associadas às mensagens. O envio de uma mensagem a um objeto acarreta a procura de um método cujo seletor seja adequado à mensagem. Se o método não for encontrado na interface da classe deste objeto a procura passa a ser feita na interface da sua superclasse, e assim sucessivamente.

Desta forma, a ligação (“binding”) da mensagem com um método a ser executado é feita em tempo de execução. Outra característica da linguagem é que não se trabalha com a verificação de tipos, já que não existe a associação de uma variável a um “tipo”.

As classes descrevem a implementação de objetos que representam componentes do sistema possuindo comportamentos idênticos. Como todas as entidades do sistema Smalltalk são vistas como objetos e todos os objetos são instâncias de uma classe, as próprias classes devem ser representadas por instâncias de uma classe, surgindo o conceito de *metaclasses*. Em versões preliminares da linguagem existia apenas uma única metaclasse, chamada Class. Com esta abordagem, o protocolo de todas as classes estava limitado a ser o mesmo. Assim, múltiplas metaclasses são necessárias porque mensagens para iniciação de novas instâncias são diferentes para diversas classes. Portanto, uma classe é uma instância única de uma metaclasse, que possui o objetivo de definir o protocolo de utilização da classe.

Todas as metaclasses são instâncias de uma classe chamada Metaclass e possuem uma superclasse abstrata (Class) que descreve a natureza geral das metaclasses. Cada metaclasse pode adicionar novas mensagens de criação de instâncias ou redefinir as mensagens fundamentais, isto é, cada metaclasse adiciona o comportamento específico à sua única instância através dos métodos da classe apresentados como parte da descrição da classe. O uso de mensagens de classe também é importante para a iniciação de variáveis de classe.

Um objeto é alocado explicitamente por intermédio da mensagem new. No entanto, não existem mecanismos para especificação de destruição de um objeto. Faz-se a utilização de uma técnica de recuperação automática de memória “garbage collection”, tornando o processo de destruição de objetos transparente ao usuário.

O mecanismo de herança é simples, uma vez que uma subclasse é derivada de apenas uma superclasse direta e muitas classes podem compartilhar a mesma superclasse, resultando em uma hierarquia em forma de árvore. O ambiente Smalltalk possui uma biblioteca de classes contendo uma hierarquia concebida na definição do sistema. A esta biblioteca novas classes podem ser adicionadas pelos usuários. Desta maneira, o programador Smalltalk sempre cria uma nova classe como sendo uma subclasse de uma classe já existente. Uma classe do sistema, chamada Object e que é a única no sistema a não possuir superclasse, descreve as similaridades de todos os objetos do sistema. Portanto, toda classe deverá pelo menos ser subclasse de Object.

Um programa em SmallTalk é uma expressão ou uma seqüência de expressões separadas pelo símbolo “**.**” (ponto). Expressões podem ser:

1. um *literal* (objeto constante como números, caracteres, cadeias e vetores compostos por literais);
2. uma *variável* (referência a um objeto);
3. um envio de *mensagem*;
4. um *bloco* (uma seqüência, delimitada por colchetes, de expressões separadas por “**.**”, cuja ação é postergada).

Na Figura 4.2. pode-se ver um trecho de programa em SmallTalk onde são definidas as classes “Empregado” e “Gerente”.

```
class name
        Empregado

superclass
        Object

Instance variable names
        nome
        salario

class variable names
        NumeroDeEmpregados

class methods
        initialize
                NumeroDeEmpregados := 0

        new
                | newEmp |
                newEmp := super new.
                NumeroDeEmpregados := NumeroDeEmpregados + 1.
                ^newEmp

instance methods
        atualizaNome: umNome
                nome := umNome

        nome
                ^nome

        atualizaSalario: umSalario
                salario := umSalario

        salario
                ^salario


class name
        Gerente

superclass
        Empregado

Instance variable names
        comissao
```

```
instance methods

        atualizaComissao: umaComissao
                comissao := umaComissao

        salario
                ^(salario + comissao)
```

Figura 4.2. Definição de classes em SmallTalk

Na Figura 4.3 está listado um trecho de programa que retorna a quantidade de números ímpar em um vetor.

```
| v nOdd |
        v := #(1 5 2 3 6).
        nOdd := 0.
        v do: [:elem | elem odd ifTrue: [nOdd := nOdd + 1 ]].
        v at: 1 put: nOdd.
        ^nOdd
```

Figura 4.3. Programa exemplo em SmallTalk

4.2. A linguagem de programação C++

A linguagem de programação C++ foi desenvolvida a partir da linguagem procedimental C, acrescentando a idéia de *tipos abstratos de dados* e fornecendo o mecanismo de *herança* para proporcionar o suporte à programação orientada a objetos. As preocupações da linguagem C com as questões de eficiência do código gerado, flexibilidade e portabilidade permaneceram presentes em C++. Existe também o compromisso de compatibilidade com a linguagem C. Portanto, as construções usuais em C deverão ser tratadas pelo compilador C++.

O conceito de tipos abstratos de dados é implementado através da construção de classes. As classes especificam um conjunto de objetos (instâncias da classe) que possuem características comuns (variáveis de instância e operações). Em C++ a declaração de uma classe pode ser dividida em três seções com o objetivo de controlar a visibilidade dos elementos da classe:

- *Privada* (“private”), que limita a manipulação das variáveis de instância e operações apenas às rotinas da própria classe, ou seja, aquelas rotinas definidas na declaração da classe, exceto o caso das funções amigas, descrito mais adiante;
- *Protegida* (“protected”), que estende o acesso aos elementos da classe a todas as suas subclasses;
- *Pública* (“public”), que define a interface das instâncias da classe. Todos os elementos declarados nesta seção estão disponíveis aos clientes da classe.

> Vale salientar que, se variáveis de instância são declaradas como públicas, torna-se possível tratá-las diretamente, ou seja, não é necessária a invocação de operações para fazer acessos. Desta forma, uma variável de instância pode ser vista como uma componente de uma estrutura (struct) em C.

As variáveis de instância declaradas em uma classe possuem um tipo associado, uma vez que a declaração é feita da mesma forma como em C. O conceito de variável de classe existente na linguagem Smalltalk é implementado através da cláusula static, isto é, uma variável declarada como static torna-se disponível a todas as instâncias de uma classe, havendo um compartilhamento do mesmo dado. As operações, métodos do Smalltalk, são implementadas como funções em C, sendo permitida a sobrecarga de identificadores, isto é, pode existir a definição de duas ou mais funções com o mesmo nome, sendo a distinção feita através do tipo e/ou do número de argumentos. O acesso a objetos de uma classe pode ser restrito às funções que são declaradas como parte da classe do objeto. Tais funções são chamadas de funções membros. Além de funções membros, funções declaradas como amigas da classe estão autorizadas a manipular elementos privados e protegidos da classe sem pertencer ao escopo desta classe. Também existe a possibilidade de fazer com que todas as funções de uma classe sejam amigas de outra classe, resultando em uma relação de amizade entre classes. Com este conceito de amizade, C++ compromete a filosofia de tipos abstratos de dados, violando o encapsulamento de informação.

A criação e iniciação de objetos são feitas por intermédio de funções membros especialmente declaradas com este objetivo, denominadas *construtores*. Um objeto pode ser criado como:

- *Automático*: criado toda vez que sua declaração é encontrada na execução do programa e destruído quando o bloco em que ele ocorre é deixado;
- *Estático*: criado no início da execução do programa e somente destruído quando o programa é terminado;
- *Dinâmico*: Criado utilizando-se o operador new e destruído através do operador delete;
- *Membro*: criado como membro de outra classe ou como elemento de um vetor ou estrutura.

Funções membros existentes para a destruição de objetos são chamadas de *destruidores*. Portanto, C++ não provê nenhum mecanismo para realizar a recuperação automática de memória “garbage collection”. O acesso a um membro de uma classe é feito de forma semelhante ao acesso de elementos de uma estrutura em C. Ao contrário de Smalltalk, a ligação de uma mensagem a um método é feita em tempo de compilação, a menos que seja solicitada a ligação dinâmica, no caso do método ser declarado como **virtual**. Uma função membro declarada com virtual especifica que poderá existir uma redefinição em quaisquer das possíveis subclasses.

A linguagem C++ suporta o conceito de *herança múltipla*. Assim, mais de uma classe pode contribuir para a formação da estrutura e do comportamento das instâncias de uma classe. O programador pode definir classes que não são especializações de nenhuma outra. Classes deste tipo são chamadas *básicas*. Para uma classe que possui duas ou mais superclasses diretas a especificidade destas

superclasses na sua definição é determinada através da ordem na qual as superclasses estão relacionadas. A primeira delas é a mais específica. A questão de conflitos de nomes foi resolvida através do operador de escopo (::).

Como os tipos existentes na linguagem C foram mantidos em C++, não torna-se restrita a estruturação de dados apenas através de classes e a linguagem pode ser utilizada sem estar associada à programação orientada a objetos.

Na Figura 4.4. pode-se ver um trecho de programa em C++ onde são definidas as mesmas classes “Empregado” e “Gerente” do exemplo em SmallTalk;

```
class empregado {
        string* nome;
        float salario;
        static int numeroDeEmpregados;
public:
        empregado();
        ~empregado() { numeroDeEmpregados--; }
        void atualizaNome(string* umNome) { nome = umNome; }
        string* obtemNome() { return nome }
        void atualizaSalario(float umSalario) { salario = umSalario; }
        float obtemSalario() { return salario }
};

int empregado::numeroDeEmpregados = 0;

empregado::empregado() {
        nome = NULL;
        salario = 0.0;
        numeroDeEmpregados++;
}


class gerente : public empregado {
        float comissao;
public:
        gerente() { comissao = 0.0; }
        void atualizaComissao(float umaComissao) { comissao = umaComissao; }
        float obtemSalario();
}

float gerente::obtemSalario();
        return (document::obtemSalario() + comissao);
}
```

Figura 4.4. Definição de classes em C++

4.3. A linguagem de programação Object Pascal

A linguagem Object Pascal é uma extensão à linguagem procedimental Pascal, suportando os conceitos do paradigma de orientação a objetos. Object Pascal foi

criada por desenvolvedores da Apple Computer e em 1986 tornou-se pública como a linguagem a ser suportada pelo ambiente de desenvolvimento da família de computadores Macintosh, da Apple. Teve um grande impulso em 1995, quando a Borland lançou o Delphi, um ambiente para programação para Windows que a utiliza como linguagem de programação para o desenvolvimento de aplicações.

Object Pascal suporta completamente os conceitos de POO. Permite a definição de objetos, o gerenciamento de objetos por classes e o relacionamento de classes através do mecanismo de herança. Suporta também a definição de métodos polimórficos, mas não permite a herança múltipla (como em SmallTalk e ao contrário de C++). Além disso, não é permitido polimorfismo paramétrico (o mesmo nome do método, diferenciado apenas pelo tipo ou quantidade de parâmetros passados na mensagem).

Além do mecanismo básico de encapsulamento obtido através dos objetos, Object Pascal suporta o encapsulamento em outro nível, através de *units*. *Units* são módulos dentre dos quais pode-se definir quaisquer elementos de Object Pascal, como variáveis, objetos, procedimentos, exceto outra *unit*. Uma unit implementa o ocultamento de informação em dois níveis de visibilidade: interface e implementação (*implementation*). Na interface são declarados os dados, procedimentos e funções visíveis fora da *unit*. Na implementação são declaradas as características visíveis somente no interior da *unit*.

É importante frisar que Object Pascal é uma linguagem de propósito geral, podendo ser empregada para desenvolver os mais variados tipos de aplicações, desde sistemas em tempo real que controlam uma indústria, até um sistema de informações de uma loja de departamentos. O que faz a diferença são as bibliotecas de classes disponíveis para um certo domínio de aplicação. No ambiente Delphi, as bibliotecas de classes para Object Pascal estão direcionadas para o desenvolvimento de sistemas de informação em arquiteturas cliente/servidor.

Em Object Pascal todos as classes são descendentes da classe *TObject*, de modo que as funções básicas de manipulação de objetos são tratadas nessa classe genérica (como o construtor e destrutor das instâncias). Se em uma declaração de classe for omitida a superclasse, automaticamente ela será subclasse de *TObject*.

Os componentes de uma classe são:

- *Campos*: definem os itens de dados que existem em cada instância da classe. Isto é similar a um campo de um registro (estrutura composta).
- *Métodos*: é um procedimento ou função que efetua alguma operação em um objeto. Existem métodos especiais que são o *construtor* e o *destrutor*. O construtor define as ações associadas com a criação de um objeto (inicializações, alocação de memória, etc.). O destrutor descreve as ações a serem tomadas quando o objeto é destruído, com a liberação de memória.
- *Propriedades*: são atributos para os quais são definidas ações para a sua leitura e escrita.

O ocultamento de informação nas classes de Object Pascal é implementado em cinco níveis, utilizando os seguintes atributos de visibilidade:

- *Public*: os componentes declarados na seção *public* de uma classe não têm nenhuma restrição quanto à sua visibilidade
- *Published*: as regras de visibilidade são idênticas às de *public*. A diferença está no fato de que é gerado informação de tipagem em tempo de execução para campos e propriedades da seção *published*.
- *Automated*: as regras de visibilidade são idênticas às de *public*. A diferença está no fato de que é gerado informação de tipagem para automação para métodos e propriedades da seção *automated*. Esta informação torna possível a criação de servidor OLE (Object Linking and Embedding).
- *Protected*: os componentes declarados na seção *protected* de uma classe somente podem ser vistos dentro da própria classe e suas descendentes.
- *Private*: a visibilidade dos componentes declarados na seção *private* de uma classe é restrita ao módulo (*unit*) que contem a declaração da classe. Ou seja, em todo o escopo da *unit*, pode-se acessar os componentes declarados na seção *private*.

Programas escritos em Object Pascal podem seguir a sintaxe e semântica tradicional da programação procedimental em Pascal, ou seja, o seu compilador, compila programas que não utilizem as técnicas de POO. A grande vantagem dos ambientes que fornecem bibliotecas de classes pré-definidas está no fato de que se obtém os benefícios da POO sem utilizá-la conscientemente. O ambiente é customizado e a programação é auxiliada pelas técnicas de POO embora o programador não se dê conta disso. O Delphi utiliza este conceito.

```
unit Empregado;

interface

type
  TEmpregado = class
    protected
      Nome: string[30];
      Salario: real;
    public
      constructor Create;
      destructor Destroy;
      procedure AtualizaNome (umNome: string);
      function ObtemNome: string;
      procedure AtualizaSalario (umSalario: real);
      function ObtemSalario: real;
      class procedure Inicializa;
    end;

  TGerente = class (TEmpregado)
    private
      Comissao: real;
    public
      constructor Create;
      procedure AtualizaComissao (umaComissao: real);
      function ObtemSalario: real;
```

```
  end;

implementation

  var
    NumeroDeEmpregados: integer;

    class procedure TEmpregado.Inicializa;
    begin
      NumeroDeEmpregados := 0;
    end;

  constructor TEmpregado.Create;
  begin
    inherited Create;
    Nome := '';
    Salario := 0.0;
    NumeroDeEmpregados := NumeroDeEmpregados + 1;
  end;

  destructor TEmpregado.Destroy;
  begin
    inherited Destroy;
    NumeroDeEmpregados := NumeroDeEmpregados - 1;
  end;

  procedure TEmpregado.AtualizaNome (umNome: string);
  begin
    Nome := umNome;
  end;

  function TEmpregado.ObtemNome: string;
  begin
    ObtemNome := Nome;
  end;

  procedure TEmpregado.AtualizaSalario (umSalario: real);
  begin
    Salario := umSalario;
  end;

  function TEmpregado.ObtemSalario: real;
  begin
    ObtemSalario := Salario;
  end;

  constructor TGerente.Create;
  begin
    inherited Create;
    Comissao := 0.0;
  end;

  procedure TGerente.AtualizaComissao (umaComissao: real);
  begin
```

```
    Comissao := umaComissao;
  end;

  function TGerente.ObtemSalario: real;
  begin
    ObtemSalario := Salario + Comissao;
  end;

end.
```

Figura 4.5. Definição de classes em Object Pascal

5. O paradigma de orientação a objetos

O paradigma de orientação a objetos apresenta a característica fundamental de unificar os esforços no desenvolvimento de sistemas. Nas seções acima foram abordados os aspectos teóricos da orientação a objetos, com ênfase na programação. Aqui será abordado o assunto de maneira mais ampla, apresentando como os vários níveis de técnicas, modelos e ambientes estão evoluindo em torno da orientação a objetos.

5.1. Desenvolvimento Orientado a Objetos

O desenvolvimento de sistemas orientado a objetos envolve a utilização de metodologias bem definidas e sedimentadas. A metodologia consiste na construção de um modelo de um domínio de aplicação e na posterior adição a este dos detalhes de implementação durante o projeto de um sistema.

Uma Metodologia Orientada a Objeto (MOO) para desenvolvimento de sistemas, consiste basicamente de: **notações gráficas**, **processos** e **ferramentas**. Além disso uma boa metodologia deveria prover (nem todas abordam os aspectos abaixo):

- Orientação para estimativa de custos;
- Aspectos de gerenciamento de projetos;
- Medidas para controle correticidade
- Procedimentos e políticas para garantia de qualidade de software
- Descrições de papéis (funções desempenhadas por pessoas) e programas de treinamento detalhados
- Exemplos completos (trabalhados no mundo real)
- Exercícios de treinamento;
- Técnicas bem definidas para utilização do método

Existem atualmente algumas Metodologias Orientadas a Objetos que podem ser consultadas em livros ou publicações especializadas, entre as quais pode-se citar:

- Metodologia de **Berard**, consistindo de:

Berard, E. V., “Essays on Object-Oriented Software Engineering”, v. 1, *Prentice Hall*, 1993.

Berard, E. V., “A Project Management Handbook for Object-Oriented Software Development”, *Berard Software Engineering*, Inc., 1992.

- Metodologia de **Booch**, consistindo de:

Booch, G., “Object-Oriented Design With Applications”, Benjamin/Cummings, 1991.

- Metodologia de **Colbert**, consistindo de:

Colbert, E., "The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development," Anais de TRI-Ada '89, ACM, out. 1989.

- Metodologia de **Coad e Yourdon**, consistindo de:

Coad, P. & Yourdon, E, “Object-Oriented Analysis”, Prentice Hall, 1990.

Coad, P. & Yourdon, E, “Object-Oriented Design”, Prentice Hall, 1990.

- Metodologia de **Embley e Kurtz**, consistindo de:

Embley, D.W. et al., “Object-Oriented Systems Analysis, A Model-Driven Approach, Yourdon Press/Prentice Hall, 1992.

- Metodologia da **IBM**, consistindo de:

IBM, “Object-Oriented Design: A Preliminary Approach” - Documento GG24-3647-00, IBM International Technical Support Center, 1990.

IBM, “Object-Oriented Analysis of the ITSO Common Scenario” - Documento GG24-3566-00. IBM International Technical Support Center, 1990

- Metodologia de **Martin e Odell**, consistindo de:

Martin, J., & Odell, J.J., “Object-Oriented Analysis and Design”, Prentice Hall, 1992.

- Metodologia de **Rumbaugh**, consistindo de:

Rumbaugh, M. et al, “Object-Oriented Modeling and Design, Prentice Hall, 1991.

- Metodologia de **Shlaer e Mellor**, consistindo de:

Shlaer, S. & Mellor, S.J., “Object-Oriented Systems Analysis: Modeling the World In Data”, Yourdon Press/Prentice Hall, 1988.

Shlaer, S. & Mellor, S.J., “Object-Oriented Systems Analysis: Modeling the World In States, Yourdon Press/Prentice Hall, 1992.

- Metodologia de **Wirfs-Brock**, consistindo de:

  Wirfs-Brock, R. et al, “Designing Object-Oriented Software”, Prentice Hall, 1990.

Na seção 5.4 será apresentada resumidamente a metodologia de Rumbaugh, conhecida como Técnica de Modelagem de Objetos (TMO).

5.2. Análise Orientada a Objetos

A análise é responsável pelo desenvolvimento de uma representação, a mais completa, concisa, correta e legível possível do domínio do problema. O analista parte do enunciado do problema, e constrói um modelo da situação do mundo real, mostrando suas propriedades relevantes. O analista deve trabalhar em conjunto com o usuário para entender o problema uma vez que as definições dos problemas raramente são completas ou corretas.

O modelo de análise é uma abstração concisa e precisa do *que* o sistema desejado deverá fazer, não *como* o sistema deverá fazê-lo. Os objetos do modelo devem ser conceitos do domínio da aplicação e não conceitos de implementação computacional, como, por exemplo, estruturas de dados. Um bom modelo pode ser compreendido e criticado por peritos em aplicações que não sejam programadores. O modelo de análise não deve conter quaisquer decisões de implementação.

A Análise Orientada a Objetos (AOO) empenha-se em entender e modelar, em termos de conceitos de orientação a objetos (objetos, classes, herança) um determinado problema dentro de um domínio de problema. Isto dentro de uma perspectiva do usuário ou de um especialista sobre o domínio em questão e com ênfase em modelar o mundo real. O produto, ou modelo resultante, da AOO especifica um sistema completo e um conjunto integral de requisitos (especificações) e a interface externa do sistema a ser construído.

Métodos convencionais de análise de sistemas (DeMarco, Yourdon, Gane & Sarson, Shlaer & Mellor) distanciam-se da forma de pensar das pessoas. Já os orientados a objetos facilitam a interação analista/usuário, pois modelam o mundo identificando classes e objetos que formam o vocabulário do domínio do problema. Como resultado, é possível a modelagem de domínios de problemas mais desafiadores com resultados de análise mais consistentes.

A análise orientada a objetos facilita a representação explícita dos pontos comuns do domínio da aplicação, através do uso de mecanismos de estruturação (herança, composição, classificação). São elaboradas especificações que facilitam as alterações que se fazem constantemente necessárias devido a mudanças de requisitos. Existe também o potencial para reutilização de resultados obtidos em análises anteriores, principalmente num mesmo domínio de aplicação. Isto leva a uma melhor qualidade dos sistemas desenvolvidos. O resultado de um processo de AOO facilita o entendimento do sistema para a etapa de projeto.

5.3. Projeto Orientado a Objetos

A etapa de projeto é o processo de mapear os requisitos de sistema definidos durante a etapa de análise para uma representação abstrata de uma implementação específica do sistema, dentro de restrições de custo e desempenho. O projetista de sistemas toma decisões de alto nível relativamente à arquitetura geral. Durante o andamento do projeto, o sistema alvo é organizado em subsistemas baseados tanto na estrutura da análise quanto na arquitetura proposta. O projetista de sistemas deve decidir quais características de desempenho devem ser otimizadas, escolher a estratégia de ataque ao problema e realizar alocações experimentais de recursos.

Projeto Orientado a Objetos (PrOO) é um método de projeto que incorpora um processo de decomposição orientado a objetos e uma notação para descrever um modelo lógico e físico, bem como estático e dinâmico do sistema projetado. Existem duas partes importantes nesta definição:

- O PrOO leva a uma decomposição orientada a objetos do sistema
- Utiliza uma notação para expressar os diferentes modelos dos projeto lógico e físico do sistema

É o suporte para decomposição orientada a objetos que torna o projeto orientado a objetos bastante diferente do projeto estruturado. Aquele usa classes e objetos como abstrações para estruturar logicamente os sistemas, enquanto este usa abstrações de funções para estruturação.

As metodologias orientadas a objetos utilizam uma progressão de AOO para PrOO para POO. Um domínio de problema tem muitas possíveis “realizações”, ou diferentes AOOs (depende de quem realiza a análise). Uma AOO tem muitas possíveis “realizações”, ou diferentes PrOOs, mas uma notação similar é utilizada para análise e projeto. Um PrOO tem também muitas realizações, ou diferentes POOs, mas permite que haja uma seleção de ma linguagem específica entre várias (ou seja, escolher a melhor linguagem para implementar o projeto). A maneira como a progressão é encaminhada e como são escolhidas as realizações depende da metodologia utilizada e do contexto com o qual a equipe de desenvolvimento tem que trabalhar.

A diferença entre análise e projeto pode, às vezes, parecer arbitrária e confusa. Mas, existem algumas regras simples que podem auxiliar a nas decisões com relação ao escopo correto da análise e do projeto.

O modelo da análise deve incluir informações significativas sob a perspectiva do mundo real e apresentar a visão externa do sistema. O modelo da análise deve ser compreensível para o cliente do sistema, devendo fornecer uma base útil para evidenciar os verdadeiros requisitos de um sistema. Os requisitos verdadeiros são aqueles realmente necessários, internamente consistentes e possíveis de serem atingidos.

Em contraste, o modelo do projeto depende da relevância para implementação em computador. Assim, ele deve ser razoavelmente eficiente e de codificação prática. Na prática, muitas partes do modelo da análise podem, freqüentemente, ser implementados prontamente, sem necessidade de modificações; desse modo pode haver uma considerável sobreposição dos modelos de análise e de projeto. O modelo de projeto deve mencionar detalhes de níveis inferiores que são omitidos no modelo

de análise. Os dois modelos combinam-se para fornecer uma valiosa documentação de um sistema, sob duas perspectivas diferentes, porém complementares.

5.4. Técnica de Modelagem de Objetos

A Técnica de Modelagem de Objetos (TMO) é uma metodologia proposta por Rumbaugh onde o processo de desenvolvimento se baseia na construção de três modelos (Modelo de Objetos, Modelo Dinâmico e Modelo Funcional), que descrevem o sistema em estudo, oferecendo três visões diferentes e complementares do mesmo. Está baseada na modelagem de objetos do mundo real e independente da linguagem de programação utilizada na implementação. A metodologia tem as seguintes etapas: análise, projeto do sistema, projeto dos objetos e implementação.

5.4.1. Análise

O objetivo da análise é desenvolver um modelo do que o sistema irá fazer. Este modelo é expresso em termos de objetos e relacionamentos, fluxo de controle dinâmico e transformações funcionais. O processo de obtenção dos requisitos e de discussões com o usuário solicitante deve manter-se durante toda a análise.

Neste fase, a entrada é a própria descrição do problema e a saída é um modelo formal do mesmo. Os passos para realizar a análise são os seguintes:

- Escrever ou obter uma descrição inicial do problema
- Construir um Modelo de Objetos
- Desenvolver um Modelo Dinâmico
- Construir um Modelo Funcional
- Verificar, repetir e refinar os três modelos

O Modelo de Objetos

O *modelo de objetos* descreve a estrutura de objetos de um sistema - sua identidade, seus relacionamentos com outros objetos, seus atributos e suas operações. É o mais importante dos três modelos, pois enfatiza a construção da arquitetura lógica do sistema, tendo como base os objetos e seus relacionamentos, que em geral são as partes mais estáveis de um sistema. O modelo de objetos proporciona a estrutura necessária sobre a qual podem ser colocados os modelos dinâmico e funcional.

A notação utilizada para descrição deste modelo é, na verdade, uma extensão da notação empregada no Modelo Entidade-Relacionamento (MER), pois combina os conceitos de orientação a objetos (classes e herança) e da modelagem da informação (entidades e associações). Possui as seguintes características:

- Utiliza grafos cujos nós representam classes de objetos e os arcos representam relacionamentos.
- As classes são organizadas em hierarquias. Elas definem os valores dos atributos carregados por cada objeto e definem as operações que cada objeto faz ou recebe.

- Proporciona uma estrutura de dados, onde o modelo funcional e dinâmico podem ser alocados
- Possui uma estrutura estática

O Modelo de Dinâmico

Descreve os aspectos temporais e comportamentais do sistema, capturando o controle e o seqüenciamento de operações. Deve definir quais são os estímulos enviados aos objetos e quais as suas respostas. Suas características são as seguintes:

- Utiliza diagramas de transição de estados (DTE). Estes diagramas são grafos, onde os nós são *estados* e os arcos dirigidos representam *transições* entre estados, sendo causados por *eventos*.
- É formado por uma coleção de DTEs que interagem uns com os outros.
- A representação gráfica mostra o estado e a seqüência de transições de estados válidas no sistema para uma classe de objetos.
- Possui uma estrutura de controle.

O Modelo Funcional

Descreve os aspectos de transformação dos dados dentro do sistema. Deve determinar quais são as computações que cada objeto executa através de suas operações. Seu objetivo e características são o seguinte:

- Identificar os valores de entrada e saída da declaração do problema e mostrar as dependências entre os valores de entrada e a computação necessária para gerar a saída desejada.
- Utiliza diagramas de fluxo de dados (DFD).
- Preocupa-se com o *que* o sistema faz, sem se preocupar *como* ou *quando* faz.
- Possui um estrutura computacional.
- Fornece uma descrição de cada função.
- Identifica restrições entre objetos e especifica critérios de otimização.

5.4.2. Projeto do sistema

Durante o projeto do sistema é determinada a estrutura de alto nível do sistema. Várias arquiteturas canônicas podem ser utilizadas como ponto de partida. O paradigma de orientação a objetos não apresenta novidades especiais em termos de projeto de sistemas. Algumas características são:

- Organizar o sistema em subsistemas
- Identificar concorrências inerentes ao problema
- Alocar subsistemas a processadores e tarefas
- Escolher a estratégia básica para implementação dos depósitos de dados em termos de estruturas de dados, arquivos e bancos de dados.
- Identificar os recursos globais e determinar mecanismos para controlar o acesso a eles.
- Considerar condições externas e estabelecer prioridades

5.4.3. Projeto dos Objetos

O projeto de objetos é um modelo baseado no modelo da análise, mas contendo detalhes de implementação. O projetista acrescenta detalhes ao modelo do projeto de acordo com a estratégia estabelecida durante o projeto do sistema. O enfoque do projeto de objetos são as estruturas de dados e os algoritmos necessários à implementação de cada classe.

As classes de objetos provenientes da análise ainda são significativas, mas são acrescidas das estruturas de dados do domínio do computador e algoritmos escolhidos para otimizar medidas importantes de desempenho. No projeto de objetos é ainda ajustada a estrutura de classes para aperfeiçoar a herança; concretizado o projeto e implementação das associações; determinada a representação exata dos atributos dos objetos e arrumadas as classes e associações em módulos.

5.5. Bancos de Dados Orientados a Objetos

Bancos de Dados Orientados a objetos (BDOO) são bancos de dados que suportam objetos e classes. Eles são diferentes dos bancos de dados relacionais mais tradicionais porque permitem uma estruturação com sub-objetos, cada objeto tendo a sua própria identidade, ou *identificador de objeto* (ao contrário do modelo relacional, onde a abordagem é orientado a valores). É possível também proporcionar operações relacionais em um BDOO. BDOOs permitem todos os benefícios da orientação a objetos, assim como têm uma forte equivalência com os programas orientados a objetos. Esta equivalência é perdida se outra alternativa for escolhido, como um banco de dados puramente relacional.

Um BDOO deve satisfazer, no mínimo, aos seguintes requisitos:

- Deve prover *funcionalidade de banco de dados*. Como tal, podem ser apresentadas: modelo e linguagem não-trivias, ou seja deve entender alguma estrutura nos dados e prover uma linguagem para manipular esses dados; relacionamentos entre os objetos; persistência, ou seja os dados devem poder ser acessados após o término do processo que os criou; compartilhamento; tamanho arbitrário; entre outras
- Deve suportar *identidade de objetos*, através do identificador de objetos, que é utilizado para localizar com precisão um objeto a ser recuperado.
- Deve prover *encapsulamento*, que é a base sobre a qual objetos abstratos são definidos.
- Deve suportar objetos com um *estado complexo*. O estado de um objeto pode se referir a outros objetos, que por sua vez podem ter referências a outros objetos, e assim por diante.
- Deve suportar *transações longas*, que podem durar horas ou dias, envolvendo um ou mais usuários (por exemplo, a edição de trechos de programa em um ambiente multiusuário).

Com relação à inclusão da herança os conceitos de BDOO, existem muitas divergências, pois sua implementação apresenta alguns problemas não triviais.

Embora a herança seja considerada a contribuição original do paradigma de orientação a objetos, algumas linhas de pensamento preferem retirá-la dos BDOOs.

BDOOs devem ser utilizados como repositório final de dados em uma metodologia de desenvolvimento que utiliza a AOO, o PrOO e a POO. Embora já existam BDOOs comercialmente disponíveis, as opções são poucas e a tecnologia não está ainda sedimentada. Por este motivo, não são utilizados em grande escala, reduzindo-se a universidades e ambientes de pesquisa. Este fato traz problemas à utilização de uma metodologia totalmente orientada a objetos, pois no momento do armazenamento dos dados é necessária uma conversão (para o modelo relacional, por exemplo), onde é perdida grande parte dos benefícios do paradigma.

Existem algumas aplicações, no entanto, que se beneficiam sobremaneira da utilização de BDOO, principalmente aquelas cujos dados não são facilmente estruturados em tabelas, como aplicações CAD (Computer-Aided Design - Projeto Auxiliado por Computador), CASE (Computer-Aided Software Enginering) e também os Sistemas de Automacão de Escritórios, entre outras. Essas aplicações de projeto impõem uma demanda muita alta nas tecnologias convencionais de bancos de dados., incluindo a capacidade de modelar dados muito complexos e a capacidade de evolução da base de aplicação sem efeitos negativos. Essa demanda por sua vez impõem requisitos no sistema de prover um nível apropriado de escalabilidade das aplicações (ver vantagens da orientação a objetos, seção 3.1) para capturar facilmente semânticas de dados específicas e mecanismos para o desenvolvimento incremental de estruturas de bancos de dados.

5.6. Outras utilizações da Orientação a Objetos

Como um paradigma unificador, várias vertentes de computação estão convergindo para a orientação a objetos, em busca dos benefícios já citados.

5.6.1. Reengenharia de Processos Orientada a Objetos

A Reengenharia de Processos Orientada a Objetos (RPOO) procura redesenhar os processos básicos da empresa com ênfase em objetos, identificando os domínios de aplicação e as principais classes de objetos e seus relacionamentos. Uma vez que a empresa como um todo está modelada de acordo com o paradigma de orientação a objetos, o desenvolvimento de sistemas torna-se uma continuação de um processo muito mais amplo, que se iniciou na própria concepção da empresa. A grande vantagem é a visão da empresa inteira como objetos que se comunicam através de mensagens.

Nestes casos, o processo de identificação dos objetos básicos de um sistema executada na fase de AOO torna-se mais rápido, conciso e confiável. A RPOO pose ser vista como uma etapa inicial na progressão determinada pela metodologia orientada a objetos, antecedendo a AOO.

Uma das experiências mais interessantes da utilização da RPOO no Brasil é a vivida pelo Banco Bamerindus, que investiu uma grande quantidade de tempo e dinheiro envolvendo pessoal extremamente qualificado do país e do exterior para reprojetar completamente as suas atividades. A partir a modelagem obtida na RPOO

foi empregada uma metodologia de desenvolvimento orientada a objetos, em conjunto com todas as tecnologias disponíveis atualmente para este tipo de aplicação.

5.6.2 Ambientes de Desenvolvimento de Software Orientados a Objetos

A definição de um Ambiente de Desenvolvimento de Software (ADS) pode ser utilizada para representar várias ferramentas de apoio ao desenvolvimento de sistemas, desde ferramentas básicas de programação até sistemas CASE integrados. Em geral, um ADS pode ser definido como uma coleção de ferramentas integradas segundo um esquema extensível, destinadas a suportar tarefas técnicas e administrativas de desenvolvimento de software em larga escala, envolvendo a interação de pessoas e grupos não necessariamente próximos, segundo um método de desenvolvimento de software que componha um ciclo qualquer completo, desde a definição de requisitos até a manutenção.

Em geral, é difícil encontrar ambientes completos que atendam aos requisitos da definição apresentada. Existem alguns esforços em áreas dispersas, como é o caso dos ambientes de programação orientados a objetos. O Delphi implementa tal ambiente, que utiliza a linguagem Object Pascal para dar suporte a um ambiente facilmente amigável e extensível devido a orientação a objetos. O Delphi implementa uma biblioteca de classes para o Object Pascal para as tarefas de programação visual em Windows, permitindo que o ambiente seja customizado dentro dele mesmo, com os seus próprios recursos (leia-se herança).

5.6.3 Interfaces Gráficas de Usuário Orientadas a Objetos

As interfaces de usuário muito se beneficiaram da orientação a objetos. Muitos sistemas de informação (aplicativos) e até sistemas operacionais oferecem atualmente interfaces que aos olhos do usuário são orientadas a objetos, que obviamente são gráficas, formando então uma GUI (Graphical User Interface). O usuário seleciona um objeto (uma janela, um ícone, etc.) e então dispõe de várias operações que o objeto suporta. A orientação a objetos está presente também (e talvez de forma mais importante) na implementação da interface contribuindo de maneira significativa para a redução dos problemas devidos a construção de GUIs.

5.6.4 Sistemas Operacionais Orientados a Objetos

Os Sistemas Operacionais Orientados a Objetos provêm recursos básicos através de objetos. Algumas vezes chegando até a gerenciar a máquina em si sob este paradigma, no caso de arquiteturas orientadas a objetos (como a projeto SOAR - SmallTalk On a Risc). Eles são quase sempre sistemas distribuídos, permitindo que objetos viajem livremente entre as máquinas. Tais sistemas são tipicamente baseados em permissões (capabilities) desde que os objetos, e dessa forma os recursos do sistema, somente podem ser acessados se os programas possuem as devidas permissões. Existem atualmente vários sistemas operacionais orientados a objetos, como Chorus, Choises, GEOS, Mach, SOS e Spring (Sun).

6. Bibliografia

Coad, P. & Yourdon, E., “Object-Oriented Analysis”, Prentice Hall, 1990.

Hathaway, B., “Object-Orientation FAQ (Frequently-Asked Question)”, *World-Wide Web: (http://iamwww.unibe.ch)*, 1996.

Kamienski, F. V., “Programação Orientada a Objetos”, Relatório Técnico, IMECC-Unicamp, fev. 1992.

Korson, T. & McGregor, J. D., “Understanding Object-Oriented: A Unifying Paradigm”, Communications of the ACM, v. 33, n. 9, set. 1990.

Martin, J. “Princípios de Análise e Projeto Baseados em Objetos”, Editora Campus, 1994.

Motta, G. H. M. B., Introdução à Orientação a Objetos: Programação, Projeto, Análise e Bancos de Dados”, material particular, 1995.

Motta, G. H. M. B., Programação Orientada a Objetos com Delphi”, material particular, 1995.

Rumbaugh, J. et al, “Modelagem e Projetos Baseados a Objetos”, Editora Campus, 1994.

Sebesta, R. W., “Concepts of Programming Languages”, Benjamin/Cummings, 1989.

Stroustrup, B., “The C++ Programming Language”, Addison-Wesley, 1986.

Takahashi, T., Liesenberg, H. K. E. & Xavier, D. T., “Programação Orientada a Objetos”, VII Escola de Computação, São Paulo, 1990.

The Object Agency, Inc., “A Comparison of Object-Oriented Development Methodologies”, *World-Wide Web: (http://www.toa.com)*, 1995.

Todd, B. & Kellen, V., “Delphi: A Developer’s Guide”, M&T Books, 1995.

Wegner, P., “Concepts and Paradigms of Object Oriented Programming”, ACM Object-Oriented Messenger, v. 1, n. 1, 1990.

Zdonik, S. B. & Maier, D., “Readings in Object-Oriented Database Systems”, Morgan Kaufmann, 1990.

