

Silvio Yochio Yamaguti

**ORIENTAÇÃO A OBJETOS NO DESENVOLVIMENTO DE SISTEMAS:  
CONCEITOS E CARACTERÍSTICAS**

Monografia final do curso de  
Engenharia de Sistemas, Pós-  
Graduação Lato Sensu, da  
Escola Superior Aberta do  
Brasil - ESAB

**Escola Superior Aberta do Brasil – ESAB  
Brasília/DF – 2006**

## Sumário

1.Introdução.....	5
1.1.Histórico.....	6
1.2.Apresentação do Modelo.....	7
2.A filosofia da Orientação a Objetos.....	9
2.1.Conceitos Básicos.....	9
2.2.Conceitos de Orientação a Objetos.....	11
2.3.Terminologia.....	12
2.3.1.Objetos.....	13
2.3.2.Mensagens.....	14
2.3.3.Classes.....	14
2.3.4.Métodos.....	15
2.3.5.Encapsulamento.....	16
2.3.6.Hierarquia.....	16
2.3.6.1.Herança.....	16
2.3.6.2. Herança Múltipla.....	17
2.3.6.3.Classe Abstrata.....	18
2.3.6.4.Agregação.....	18
2.3.7.Polimorfismo.....	19
2.3.8.Reutilização.....	19
2.3.9. Associação.....	20
3.Ciclo de Vida.....	21
3.1.Análise.....	21
3.2.Projeto do Sistema.....	23
3.3.Projeto de Objetos.....	24
3.3.1.O Modelo de Objetos.....	25
3.3.2.O Modelo Dinâmico.....	25
3.3.3.O Modelo Funcional.....	26
4.Metodologias Orientadas a Objetos.....	28
4.1.Metodologia de Booch.....	28
4.1.1.Conceitos.....	29
4.1.2.Enquadramento Conceitual.....	29
4.1.3.Processos e técnicas.....	30
4.1.4.Ferramentas de suporte.....	32
4.2.Metodologia de Rumbaugh.....	33
4.2.1.Conceitos.....	34
4.2.2.Enquadramento Conceitual.....	35
4.2.3.Processos.....	35
4.2.4. Técnicas.....	37
4.2.5.Ferramentas de suporte.....	38
4.3.Metodologia de Jacobson.....	38
4.3.1.Conceitos.....	39
4.3.2. Processos.....	42
4.3.3.Técnicas.....	45
4.3.4.Ferramentas de suporte.....	45
4.4.Metodologia UML.....	46
4.4.1.Objetivos.....	46
4.4.2.Ciclo de Desenvolvimento.....	47
4.4.3.Notação da UML.....	49

<a href="#">4.4.4. Diagramas.....</a>	<a href="#">50</a>
<a href="#">5. Bancos de Dados e Linguagens Orientados a Objeto.....</a>	<a href="#">56</a>
<a href="#">5.1. Bancos de Dados Orientados a Objeto.....</a>	<a href="#">56</a>
<a href="#">5.2. Linguagens Orientadas a Objeto.....</a>	<a href="#">58</a>
<a href="#">6. Escolha da Orientação a Objeto no desenvolvimento de Sistemas.....</a>	<a href="#">61</a>
<a href="#">7. Comentários e Considerações Finais.....</a>	<a href="#">63</a>
<a href="#">8. Referências Bibliográficas.....</a>	<a href="#">67</a>



## 1. Introdução

No início da era dos computadores o custo do software era muitas vezes desprezado comparado ao custo do hardware. Com a crescente demanda por aplicações cada vez maiores e mais complexas, os custos do software aumentaram assustadoramente superando em várias vezes os custos do hardware, e a tendência é que essa progressão continue. A necessidade de desenvolvimentos mais ágeis, com ganhos em custos, em confiabilidade, em flexibilidade e em facilidade de manutenção, mesmo em ambientes complexos, que permitam um aumento sensível de produtividade, leva à adoção de novas abordagens.

É importante observar que, associada ao acréscimo da demanda, a evolução do hardware tem sido bastante acentuada, disponibilizando aos usuários máquinas cada vez mais velozes e com maior capacidade de processamento.

Neste contexto, novas metodologias de análise, desenho, modelagem, desenvolvimento e implementação foram sugeridas, documentadas, publicadas e implementadas. Buscando melhorar a qualidade dos produtos de software e visando aumentar a produtividade no processo de desenvolvimento, surgiu uma área de pesquisa denominada de Engenharia de Software, que busca organizar esforços no desenvolvimento de ferramentas, metodologias e ambientes de suporte ao desenvolvimento de software. Dentre essas ferramentas destaca-se o paradigma Orientado a Objetos.

As técnicas de Orientação a Objeto pressupõem que se possam criar sistemas compondo-se objetos previamente criados reduzindo-se o tempo de desenvolvimento, com um significativo ganho de produtividade e qualidade.

Este trabalho visa apresentar a filosofia de análise, modelagem e desenvolvimento dos Sistemas Orientados a Objetos, seus conceitos básicos, suas principais características, as principais metodologias de Orientação a Objetos e suas características técnicas, bem como os cuidados necessários para que se obtenha sucesso na utilização da Orientação a Objeto.

## 1.1. Histórico

Durante a década de 70, os analistas empregaram os conceitos de análise estruturada, que utilizava a representação gráfica para os requerimentos do sistema e sua preocupação residia no fluxo de dados. As suas limitações estavam em não estabelecer uma fronteira bem definida entre o modelo lógico e o modelo físico, ser centrada em processos, apresentar pouca preocupação com os dados, ser voltada para grandes sistemas “*mainframe*”, altamente burocrática e consumidora de tempo.

No início dos anos 80, surgiu a análise essencial para sistemas em tempo real, que ajudaram a análise estruturada a se tornar mais eficiente. Ela estabelece o particionamento por eventos, delimitando a área de atuação do sistema, através da utilização da lista de eventos, o conceito de “tecnologia perfeita” e incorpora a modelagem de dados na especificação, introduzindo o modelo Entidade Relacionamento e Atributo (ERA). As suas principais vantagens residem em oferecer critérios objetivos para o particionamento do sistema, observando suas características, facilita a obtenção da solução, através dos eventos, isola os requisitos de restrições de implementação, definindo a fronteira entre o físico e o lógico, reduz o tempo de análise e sincroniza a modelagem de dados com a de processos. Entretanto os sistemas desenvolvidos baseados nas técnicas estruturadas ainda apresentam algumas limitações, principalmente quando da inclusão de novas funções ou alterações em funções já existentes, que na maior parte das vezes, provocam problemas em outras partes do software.

A necessidade de desenvolvimentos mais ágeis, mesmo em ambientes complexos, permitindo um aumento de produtividade, levou a abordagens como a *prototipagem*, que possibilitam criar modelos rápidos e de modo dinâmico, contando com a participação efetiva do usuário. Apesar de incorporada em algumas metodologias tradicionais, parece ainda não ser a resposta definitiva ao problema.

As linguagens de modelagem Orientadas a Objetos surgiram por volta da década de 80, à medida que o pessoal envolvido com metodologia, diante de um novo gênero de linguagens de programação Orientadas a Objeto e de aplicações cada vez mais complexas, começou a experimentar métodos

alternativos de análise e projeto. A quantidade de métodos orientados a objetos aumentou significativamente durante o período final da década de 80 a meados da década de 90, levantadas por autores como Shlaer e Mellor (1990), Rumbaugh et al (1994), Booch (1996) e Jacobson et al (1992). Finalmente, no fim dos anos 90, chega-se à maturidade em Orientação a Objetos e qualidade de Software, com a utilização da UML (*Unified Modelling Language*) como linguagem padrão para modelagem.

## 1.2. Apresentação do Modelo

A Técnica de Modelagem de Objetos (TMO) segundo Rumbaugh et al (1994) compreende no mínimo, dois aspectos ortogonais ou dimensões para descrever um sistema complexo: a dimensão *estrutural dos objetos* e a dimensão *dinâmica do comportamento*, e considera também uma dimensão adicional: a dimensão *funcional dos requisitos*.

De acordo com Rumbaugh et al (1994), a dimensão estrutural dos objetos está baseada no aspecto estático ou passivo. Está relacionada com a estrutura estática dos objetos que formam o Sistema. Ela inclui a identidade de cada objeto, sua classificação, seu encapsulamento (atributos e operações) e seus relacionamentos estáticos (hierarquias de heranças, agregados, composição e associações específicas).

Ainda segundo Rumbaugh et al (1994), a dimensão dinâmica do comportamento tem a ver com o aspecto dinâmico ou ativo, por isso descreve o comportamento e a colaboração dos objetos que constituem o sistema. O comportamento é refletido através dos estados (passos dentro do ciclo de vida do objeto que caracterizam comportamentos diferentes do mesmo), transições entre esses estados, eventos (fatos que ocorrem e produzem as transições) e ações (representadas pelos métodos dos objetos, podendo ocorrer durante as transições, ou não). A colaboração é representada por modelos que mostram o fluxo de eventos ou mensagens entre os objetos. Assim, algumas ações geradas em um objeto podem levar a transições em outros objetos.

Para o caso da dimensão funcional dos requisitos, é considerado o aspecto relativo à função de transformação global do sistema, ou seja, a

conversão de entradas em saídas. Esta transformação global é refletida por processos ou funções (que transformam valores) e fluxos de dados (entradas e saídas dessas funções), configurando com isso redes funcionais (RUMBAUGH ET AL, 1994).

Esta forma é claramente oposta ao conceito de encapsulamento de métodos nos objetos, por isto existem controvérsias na literatura sobre a conveniência ou não de utilizá-la. Alguns autores sugerem o modelamento de processos denominado *end-to-end* como uma alternativa para abordar este aspecto (BAILIN,1989; JALOTE, 1989; FUCHMAN e KEMERER, 1992).

As descrições anteriores das dimensões foram feitas distintamente, porém não significa que necessariamente sejam construídos modelos separados para cada aspecto. Mais de um aspecto podem ser modelados simultaneamente, sem prejuízo dos elementos mencionados em cada dimensão.

## 2. A filosofia da Orientação a Objetos

Segundo Shlaer e Mellor (1990) a Técnica de Modelagem da Informação consiste numa organização e uma notação gráfica apropriadas para descrever e definir o vocabulário e a conceitualização do domínio de um problema. Propõe uma forma de pensar de modo abstrato, problemas a resolver empregando conceitos do mundo real ao invés de conceitos de computadores. A notação gráfica proposta ajuda no desenvolvimento de software visualizando o problema sem recorrer de forma prematura à implementação.

Ainda segundo Shlaer e Mellor (1990) o Modelo de Informação se fundamenta em pensar acerca de problemas a resolver empregando modelos que estão organizados tomados como base conceitos do mundo real. Identifica, classifica e sumariza o que está no problema; e organiza as informações numa estrutura formal. “Coisas”, ou exemplos, similares dentro do problema são identificadas e resumidas como *objetos*; as características destas instâncias são resumidas como *atributos*; e as associações confiáveis entre as instâncias são resumidas como *relacionamentos*. Este processo de sumarização requer que cada situação esteja sujeita e conforme às políticas ou às normas bem definidas e declaradas explicitamente do domínio do problema.

### 2.1. Conceitos Básicos

A Metodologia *Object Modeling Technique* – OMT vai desde a análise até a implementação passando pelo desenho ou modelagem. Em primeiro lugar, se constrói um modelo de análise para abstrair os aspectos essenciais do domínio da aplicação sem levar em conta a implementação eventual. Neste modelo tomam-se decisões importantes que depois se completam para otimizar a implementação em segundo lugar. Os objetos do domínio da aplicação constituem o marco de trabalho do modelo de desenho, porém são implementados em termos de objetos do domínio do computador. Finalmente, o modelo de desenho é implementado em uma linguagem de programação e/ou base de dados (RUMBAUGH ET AL, 1994).

Segundo Rumbaugh(1994), os objetos do domínio da aplicação e do domínio do computador podem ser modelados, desenhados e implementados utilizando os mesmos conceitos e a mesma notação orientada a objetos. Esta mesma notação é utilizada desde a análise até a implementação passando pelo desenho, de uma forma tal que uma informação acrescida em uma fase de desenvolvimento não se perde, nem necessita ser traduzida para uma próxima fase.

Ainda segundo Rumbaugh(1994), a essência do desenvolvimento Orientado a Objetos é a identificação e organização de conceitos (objetos) do domínio da aplicação. A maior parte do esforço realizado até o momento na comunidade Orientada a Objetos, tem se centrado em temas de linguagens de programação. Entretanto, em certo sentido, esta ênfase supõe um retrocesso à engenharia de software ao concentrar-se excessivamente nos mecanismos de implementação e não no processo de pensamento subjacente ao qual servem de base (Análise e Desenho). Somente quando se tem identificado, organizado e compreendido os conceitos inerentes da aplicação é que se pode passar ao tratamento efetivo dos detalhes das estruturas de dados e das funções. É uma premissa básica que os erros das primeiras fases do processo de desenvolvimento têm muita influência sobre o produto final e também sobre o tempo requerido para finalizar.

O benefício principal do desenvolvimento Orientado a Objetos não é reduzir o tempo de desenvolvimento. Ele pode levar mais tempo que o desenvolvimento convencional porque na Orientação a Objetos se pretende que haja a promoção da reutilização futura e a redução dos erros posteriores e futuras manutenções. O tempo transcorrido até que o código esteja completo pela primeira vez é possivelmente o mesmo que o transcorrido em uma codificação convencional, ou em alguns casos ligeiramente maior. O benefício da Orientação a Objetos consiste em que as interações subseqüentes são mais rápidas e mais fáceis do que empregando um método convencional, porque as revisões estão mais localizadas. A prática mostra que é necessário um menor número de interações porque um maior número de problemas são descobertos e corrigidos durante o desenvolvimento.

## 2.2. Conceitos de Orientação a Objetos

A metodologia Orientada a Objeto (OO) busca uma abordagem paralela aos métodos tradicionais de modelagem e desenho, onde o enfoque baseia-se na compreensão do sistema como um conjunto de programas que executam processos sobre dados. O enfoque em Orientação a Objetos visualiza o mundo como um conjunto de objetos que interagem entre si, apresentam características distintas representadas pelos seus atributos (dados) e operações (processos), de acordo com Coad e Yourdon (1992). Seus princípios básicos são:

- Não existe controle central;
- Não existem dados globais;
- A responsabilidade do processamento fica centrada nos objetos;
- A comunicação entre os objetos é feita exclusivamente através de mensagens.

Coad e Yourdon (1992) apresentam as principais características e benefícios dessa abordagem:

- i. Manter a modelagem do sistema, e sua automação, a mais próxima possível de uma visão conceitual do mundo real;
- ii. Permitir uma modelagem e especificações mais precisas;
- iii. Permitir uma transição sem obstáculos da fase de modelagem para a de desenho e implementação, sem necessidade de uma reorganização do modelo;
- iv. Melhorar a comunicação entre usuários e desenvolvedores;
- v. Favorecer a reutilização de código e do projeto;
- vi. Facilitar a reutilização dos modelos, minimizando o esforço e permitir especificar por extensão;
- vii. Favorecer a manutenibilidade.

Dentre as vantagens reais no nível de projeto, Coad e Yourdon (1992), citam:

- i. **Reusabilidade de código:** aumento da qualidade e produtividade no desenvolvimento de sistemas, em função do reuso;
- ii. **Ciclo de vida:** sistemas com ciclo de vida mais extenso;

- iii. **Mantenabilidade:** menor custo para manutenção de sistemas;
- iv. **Escalabilidade de aplicações:** possibilidade de produção de sistemas mais complexos, em função da possibilidade de incorporação de funções prontas (Componentes).

A abordagem Orientada a Objetos tem como principal característica analisar o mundo como ele é, e representá-lo como o vemos, isto é, representar pessoas, lugares e coisas que se comportam como objetos que possuem atributos e comportamentos próprios, permitindo organizar resultados de maneira mais simples, fácil e natural do que com modelos convencionais estruturados. A idéia do paradigma Orientado a Objetos é construir software montando peças. Segundo Rumbaugh et al (1994), Orientação a Objetos é:

*Uma nova maneira de pensar os problemas utilizando modelos organizados a partir de conceitos do mundo real. O componente fundamental é o objeto que combina estrutura e comportamento em uma única entidade.*

### 2.3. Terminologia

O propósito do modelo de objetos é capturar os conceitos existentes no domínio do problema e os relacionamentos existentes entre eles. A notação do modelo de objetos se baseia em uma extensão da notação do modelo entidade-relacionamento descrita por DeMarco (1989). Pode representar classes, atributos e relacionamentos entre classes. As extensões permitem o uso de agregação e generalização.

Existem vários conceitos que são próprios do modelo de objetos e outros inerentes à tecnologia. Ainda que nem todos sejam exclusivos dos Sistemas Orientados a Objetos, eles estão embasados por este paradigma.

### 2.3.1. Objetos

Segundo Martin (1994) objeto é “qualquer coisa, real ou abstrata, a respeito da qual armazenamos dados e os métodos que os manipulam.

Coleman et al (1996) encara os objetos como “átomos do processo de computação que trocam mensagens entre si.

Na visão de Rumbaugh et al (1994) um *objeto* é definido como um conceito, uma abstração, algo com limites nítidos e significado em relação ao problema em causa. Os objetos servem a dois objetivos: eles facilitam a compreensão do mundo real e oferecem uma base real para a implementação em computador. A decomposição de um problema em objetos depende do julgamento e da natureza do problema.

Para Shlaer e Mellor (1990), um objeto é uma abstração de um conjunto de coisas do mundo real, de forma que:

- todas as coisas do mundo real do conjunto – as instâncias – tenham as mesmas características,
- todas as instâncias estejam sujeitas a, e em conformidade com as mesmas normas.

Segundo Coleman et al (1996), um objeto representa um “elemento” que pode ser identificado de maneira única. Em um nível apropriado de abstração, praticamente tudo pode ser considerado como objeto. Assim, elementos específicos como pessoas, organizações, máquinas ou eventos podem ser considerados como objetos. Além de ser unicamente identificado, um objeto pode possuir um ou mais valores a ele associados. Em geral, um objeto pode possuir uma série de campos que armazenam valores, denominados atributos.

Ainda segundo Coleman et al (1996), os valores dos atributos de um objeto podem ser alterados, mas a quantidade e o nome dos atributos são constantes. Os atributos assumem valores de tipos primitivos, como inteiros, booleanos, textos e enumerações. Daí se segue que os objetos são mais do que apenas tuplas de valores, porque podemos ter vários objetos distintos com os mesmos valores de atributos.

### 2.3.2. Mensagens

Os objetos se comunicam através de mensagens, que consistem em um sinal enviado de um objeto para o outro, com ou sem transferência de parâmetros, solicitando algum serviço. Esta solicitação é processada no objeto destinatário e outra mensagem de resposta é enviada ou não ao objeto solicitante.

Para Martin (1994) quando um objeto envia uma mensagem, ele está acionando algum método do objeto destinatário de acordo com. Uma mensagem é formada por três componentes básicos, segundo Martin (1994):

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

Esses três componentes são suficientes para o objeto receptor executar o método desejado. Nada mais é necessário. Então, objetos em processos distintos ou ainda em máquinas distintas podem comunicar-se através do uso de mensagens (MARTIN, 1994).

### 2.3.3. Classes

Segundo Coleman et al (1996), os objetos são agrupados em conjuntos, denominados *classes*. Uma classe é uma abstração que representa a idéia ou noção geral de um conjunto de objetos similares. É um conjunto de objetos que possuem os mesmos atributos e as mesmas operações, ou seja, categoriza objetos que possuem propriedades similares, configurando-se em um modelo para a criação de novas instâncias. Associado a cada classe temos um predicado, o qual define os critérios para a inclusão de membros na classe. Assim, sempre será possível dizer se um determinado objeto pertence a uma classe em particular.

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 (RUMBAUGH ET AL, 1994).

Para Jacobson et al (1992), 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.

Ainda segundo Jacobson et al (1992), a classe define a estrutura e o comportamento dos objetos, e todos os objetos desta têm a mesma estrutura (conjunto de atributos) e comportamento (conjunto de operações) da classe a que pertencem. Cada objeto é a instância de uma classe. “Instanciar uma classe” significa criar um novo objeto da classe. 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, quaisquer diferenças de respostas a mensagens aceitas por elas, é determinada pelos valores das variáveis de instâncias.

#### **2.3.4. Métodos**

Um método implementa algum aspecto do comportamento do objeto. Comportamento ou operação 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, objeto este chamado de receptor (RUMBAUGH et al, 1994).

### **2.3.5. Encapsulamento**

Denominado também **ocultamento de informação**, consiste em separar os aspectos externos ou públicos do objeto que podem ser acessados por outros objetos, dos detalhes internos de implementação do mesmo, que estarão ocultos. Encapsular significa ocultar o estado do objeto e a implementação de suas operações. A utilização do conceito de encapsulamento faz com que só se tome conhecimento ou modifique o valor de um atributo (parte privada) através de suas operações (parte pública). Para se implementar este conceito é necessário realizar uma divisão de responsabilidades por todos os objetos, isto é, todos os objetos terão que ter atributos e métodos que o possibilite responder a todas as solicitações de outros métodos. A implementação de um objeto pode ser modificada sem afetar às aplicações que o utilizam. Talvez seja necessário modificar a implementação de um objeto para melhorar o rendimento, corrigir um erro, consolidar o código ou para migrar a uma outra plataforma (MARTIN, 1994).

### **2.3.6. Hierarquia**

Para Martin (1994) os objetos de um sistema são organizados em hierarquias, e estas no modelo de objetos devem obedecer às hierarquias correspondentes no mundo real. Os tipos de hierarquia são:

- Herança
- Agregação

#### **2.3.6.1. Herança**

Para Rumbaugh et al (1994), herança é compartilhar atributos e operações entre classes tomando como base uma relação hierárquica. É definida como a capacidade de uma nova classe em herdar atributos e métodos pertencentes a uma classe existente. Em termos gerais, pode-se definir uma classe que depois será refinada sucessivamente para produzir

subclasses. Todas as subclasses possuem ou herdam cada uma das propriedades de sua superclasse e também suas propriedades exclusivas. Não é necessário redefinir as propriedades das superclasses em cada subclasse.

Permite a criação de classes complexas sem que seja necessário repetir o código, pois a nova classe herda seu nível base de características de um antepassado na hierarquia de classe. A estrutura de herança organiza as classes em relações de generalização e especialização, isto é, subclasses especializam uma classe e superclasses generalizam uma classe.

Ainda segundo Rumbaugh et al (1994), na superclasse ficam todos os atributos que são comuns às diversas subclasses, e qualquer alteração no atributo da superclasse reflete automaticamente nas subclasses. Uma subclasse herda todas as características da superclasse, estrutura (atributos) e comportamento (operações), e ela deverá estender a estrutura e o comportamento da superclasse, criando novos atributos e criando novos métodos, respectivamente. A herança evita que se projete e implemente novamente um código já existente, configurando-se em um mecanismo para organização, construção e uso de classes reutilizáveis proporcionando característica de extensibilidade e reusabilidade ao software.

#### **2.3.6.2. Herança Múltipla**

De acordo com Jacobson et al (1992), a herança múltipla ocorre quando uma classe herda atributos e comportamentos de mais de uma classe, e consiste numa variação semântica da generalização, onde um tipo pode ter mais de um supertipo.

Ainda de acordo com Jacobson et al (1992), a herança múltipla apresenta alguns problemas como a colisão de nomes, herança duplicada, administração complexa, impactos indesejáveis na manutenção e não é suportada por algumas linguagens OO.

### **2.3.6.3. Classe Abstrata**

Para Jacobson et al (1992), classes abstratas são classes sem instâncias, ou seja, sem objetos, e que têm o propósito de prover características comuns para suas subclasses e padronizar suas interfaces (oposto: classe concreta), e além de fornecer uma modelagem mais robusta ao programador, as classes abstratas também são representadas de forma mais eficiente, pois nunca precisarão ser instanciadas.

### **2.3.6.4. Agregação**

Segundo a definição de Rumbaugh et al (1994), esta hierarquia permite representar no Modelo de Objetos estruturas “todo-parte”, onde um objeto agregado contém como atributos, os de outros objetos de sua ou de outras classes. Um relacionamento de agregação é uma associação que reflete a construção física ou a posse lógica. Relacionamentos de agregação são casos particulares dos relacionamentos de associação, e só é necessário distingui-los quando for conveniente enfatizar o caráter “todo-parte” do relacionamento. Geralmente um relacionamento de agregação é caracterizado pela presença da expressão “parte de” na descrição do relacionamento, e pela assimetria da navegação. Nesta hierarquia, diferentemente de como ocorria com a herança, todos os atributos e métodos das partes vão para o todo, descrevendo um movimento de baixo para cima, e não de cima para baixo como na herança.

Ainda segundo Rumbaugh et al (1994), um tipo mais forte de relacionamento de agregação é a composição, que determina que o objeto parte pode pertencer a somente um todo e as partes devem viver e morrer com o todo: A agregação é utilizada quando a relação todo-parte for clara no mundo real, isto é, quando as partes não podem existir sem o todo.

### **2.3.7. 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. Significa que uma operação pode comportar-se de modo distinto em distintas classes tendo o mesmo nome de método. Em termos práticos, o polimorfismo permite referir-se a objetos de diferentes classes por meio do mesmo elemento de programa e realizar a mesma operação de forma diferente, de acordo com o objeto a que se faz referência em cada momento. No mundo real uma operação é simplesmente uma abstração de comportamento análogo entre classes distintas de objetos. Objetos de classes distintas podem realizar operações conceitualmente iguais, mas com implementações diferentes. O polimorfismo é a possibilidade de dois objetos de classes diferentes possuírem duas operações com o mesmo nome e implementações diferentes, ou seja, representa as várias formas que um método pode assumir. As vantagens do polimorfismo residem em facilitar a implementação do conceito de delegação, evitar construções do tipo “case” (*if..then e else if..then*), simplificar a criação de novas subclasses e ser conceitualmente mais intuitivo. Uma outra vantagem é que uma mesma mensagem pode ser enviada para um conjunto de objetos de classes distintas, ativando vários métodos polimórficos. De maneira prática isto quer dizer que a operação em questão mantém seu comportamento transparente para quaisquer tipos de argumentos; isto é, a mesma mensagem é enviada a objetos de classes distintas e eles poderão reagir de maneiras diferentes. Um método polimórfico é aquele que pode ser aplicado a várias classes de objetos sem que haja qualquer inconveniente (JACOBSON et al, 1992).

### **2.3.8. Reutilização**

Os conceitos acima apresentados como encapsulamento, herança, polimorfismo e agregação são os que permitem concretizar a promessa de reutilização de código e extensibilidade pregada na análise de sistemas baseadas em OO.

Para Jacobson et al (1992), a herança tanto de estruturas de dados como de comportamentos, permite compartilhar uma estrutura comum entre várias subclasses similares sem redundância. Uma das principais vantagens das linguagens orientadas a objetos é compartilhar código empregando a herança, segundo.

Jacobson et al (1992) diz também que o desenvolvimento orientado a objetos permite não somente compartilhar informação dentro de uma aplicação, mas também oferece a perspectiva de se reutilizar desenhos e códigos em futuros projetos. Proporciona ferramentas tais como abstração, encapsulamento e herança para construir bibliotecas de componentes reutilizáveis. Deve-se ter em mente que a reutilização deve ser planejada pensando um pouco além da aplicação imediata, e deve-se investir um esforço adicional em se obter um desenho generalizado da aplicação.

### **2.3.9. Associação**

De acordo com Rumbaugh et al (1994), a noção de associação não é um conceito novo. As associações têm sido amplamente utilizadas na modelagem de bancos de dados por anos. Inversamente, poucas linguagens de programação suportam associações explicitamente. Ainda assim, as associações são uma útil ferramenta de modelagem para programas bem como para bancos de dados e sistemas do mundo real, independentemente da forma como sejam implementados. Durante a modelagem conceitual, não se deve colocar dentro de objetos ponteiros e outras referências a objetos como atributos. Algumas informações transcendem, por suas próprias naturezas, uma classe única, e o não tratamento das associações juntamente com as classes pode conduzir a programas contendo pressupostos ocultos e dependências.

### **3. Ciclo de Vida**

Para Rumbaugh et al (1994), o ciclo de vida completo de um software passa pela formulação inicial do problema, pela análise, projeto, implementação e pelos testes do software, e é seguido por uma fase operacional durante a qual são executados a manutenção e o aperfeiçoamento. A metodologia OMT (*Object Modelling Technique*) dá suporte a todo o ciclo de vida do software. Essa metodologia consiste das seguintes fases:

- Análise
- Projeto do Sistema
- Projeto de Objetos

#### **3.1. Análise**

Segundo Rumbagh (1994), o objetivo da análise é desenvolver um modelo do que o sistema irá fazer, ou seja, definir e permitir o entendimento do problema e do domínio da aplicação para que se possa construir um projeto correto. Esse modelo é expresso em termos de objetos e relacionamentos, fluxo de controle dinâmico e transformações funcionais. Uma análise bem feita incorpora as características essenciais do problema sem introduzir aspectos da implementação que restringem prematuramente as decisões de projeto. O processo de obtenção dos requisitos e de discussões com o usuário solicitante deve manter-se durante toda a análise.

Ainda segundo Rumbaugh et al (1994), a Análise Orientada a Objetos 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 Análise Orientada a Objetos especifica um sistema completo e um conjunto integral de requisitos (especificações) e a interface externa do sistema a ser construído. Nesta fase, a entrada é a própria descrição do problema e a saída é um modelo formal do

mesmo. Para Rumbaugh et al (1994) os passos para realizar a análise são os seguintes:

1. Escrever ou obter uma descrição inicial do problema;
2. Construir um Modelo de Objetos (diagrama do modelo de objetos e dicionário de dados):
  - Identificar as classes de objetos;
  - Iniciar a geração de um dicionário de dados contendo descrições das classes, atributos e associações
  - Acrescentar as associações entre classes;
  - Acrescentar os atributos para objetos e ligações;
  - Organizar e simplificar as classes de objetos utilizando herança;
  - Testar os caminhos de acesso utilizando roteiros e repetindo os passos anteriores, se necessário;
  - Agrupar classes em módulos, com base em enteito acoplamento e função relacionada.
3. Desenvolver um Modelo Dinâmico (diagrama de estados e diagrama global do fluxo dos eventos):
  - Preparar roteiros das seqüências típicas de interação;
  - Identificar eventos entre objetos e preparar uma seqüência de eventos para cada roteiro;
  - Preparar um diagrama de fluxo de eventos para o sistema;
  - Desenvolver um diagrama de estados para cada classe que tenha comportamento dinâmico importante;
  - Verificar a consistência dos eventos compartilhados pelos diagramas de estados.
4. Construir um Modelo Funcional (diagrama de fluxo de dados e restrições):
  - Identificar os valores de entrada e saída;
  - Utilizar diagramas de fluxo de dados quando necessário, para mostrar dependências funcionais;
  - Descrever o que cada função faz;

- Identificar as restrições;
  - Especificar os critérios de otimização.
5. Verificar, repetir e refinar os três modelos:
- Acrescentar ao modelo de objetos as operações-chave que foram descobertas durante a preparação do modelo funcional. Não apresentar todas as operações durante a análise porque poderia tornar o modelo de objetos confuso; mostrar apenas as operações mais importantes;
  - Verificar se as classes, associações, atributos e operações estão consistentes e completos no nível escolhido de abstração. Comparar os três modelos com o enunciado do problema e conhecimentos relevantes do domínio, e testar os modelos utilizando roteiros;
  - Desenvolver roteiros mais detalhados (incluindo condições de erros) como variações dos roteiros básicos. Utilizar esses roteiros hipotéticos para verificar mais detalhadamente os três modelos;
  - Repetir os passos anteriores tantas vezes quantas forem necessárias para completar a análise.

### **3.2. Projeto do Sistema**

De acordo com Rumbaugh et al (1994), projeto do sistema é onde o projetista de sistemas toma decisões de alto nível relativamente à arquitetura geral. 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. 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 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. Os passos a seguir devem fazer parte das decisões do projetista durante esta fase:

1. Organizar o sistema em subsistemas;
2. Identificar concorrências inerentes ao problema;
3. Alocar os subsistemas a processadores e tarefas;
4. Escolher a estratégia básica para implementação dos depósitos de dados em termos de estrutura de dados, arquivos e bancos de dados;
5. Identificar os recursos globais e determinar mecanismos para controlar o acesso a eles;
6. Definir uma abordagem para a implementação do controle de *softwares*:
  - Utilizar uma área dentro do programa para salvar o estado, ou
  - Implementar diretamente uma máquina de estados, ou
  - Utilizar tarefas concorrentes.
7. Considerar condições extremas;
8. Estabelecer prioridades.

### **3.3. Projeto de Objetos**

Rumbaugh et al (1994) propõe que o projeto de objetos seja 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.

A Técnica de Modelagem de Objetos é uma metodologia proposta por Rumbaugh et al (1994) 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.

### **3.3.1. O Modelo de Objetos**

Para Rumbaugh et al (1994), o modelo de objetos descreve a estrutura estática (de dados), dos objetos do sistema (identidade, atributos e operações) e também suas relações. O modelo de objetos contém diagramas de objetos. Um diagrama de objetos é uma representação gráfica cujos nós são classes de objetos e as linhas representam relações entre as classes. O diagrama contém classes de objetos organizados em hierarquias que compartilham uma estrutura e comportamentos comuns e que estão associados a outras classes. 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), e 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.

### **3.3.2. O Modelo Dinâmico**

Segundo Rumbauch (1994) o modelo dinâmico descreve os aspectos de comportamento (de controle) de um sistema que variam com o tempo, ou seja, descreve os aspectos temporais e comportamentais do sistema, capturando o controle e o seqüenciamento de operações, e deve definir quais são os estímulos enviados aos objetos e quais as suas respostas. Suas características são as seguintes:

- Utiliza diagramas de estados. Um diagrama de estados é um grafo cujos nós são estados e os arcos representam transições entre estados 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 objetos;
- Possui uma camada de controle.

As ações dos diagramas de estados são correspondentes com as funções procedentes do modelo funcional. Os eventos de um diagrama de estados passam a ser operações que se aplicam a objetos dentro do modelo de objetos.

### **3.3.3. O Modelo Funcional**

Ainda segundo Rumbaugh et al (1994) 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 suas características são:

- 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 uma estrutura computacional.
- Fornece uma descrição de cada função.

- Identifica restrições entre objetos e especifica critérios de otimização.

O modelo funcional utiliza diagramas de fluxo de dados. Um diagrama de fluxo de dados é um grafo cujos nós são processos e os arcos são fluxos de dados, onde se mostram as dependências entre os valores e o cálculo de valores de saída a partir dos de entrada e de funções sem considerar quando as funções são executadas (RUMBAUGH et al, 1994).

A complexidade das aplicações exige modelos funcionais para especificar os requisitos, porém as técnicas consideram que isto pode influir negativamente na orientabilidade dos sistemas, segundo Bailin (1989).

A questão chave parece ser o critério de particionamento do aspecto funcional. Caso o particionamento seja por decomposição funcional (estratégia *top-down* ou refinamento sucessivo) como proposto na análise estruturada tradicional de deMarco (1978), isto se torna contrário à orientação a objetos.

Caso as funções estejam subordinadas aos objetos, em cujo caso se utilizam métodos, não existiria problemas de encapsulamento e a especificação corresponderia a uma modelagem de processos *end-to-end*, de acordo com Bailin (1989).

## 4. Metodologias Orientadas 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. Existem atualmente várias Metodologias Orientadas a Objetos, mas apenas algumas serão descritas a seguir, não de forma exaustiva, mas sim apresentando uma breve descrição dos conceitos, processos, técnicas e ferramentas de suporte utilizadas em cada uma delas, de forma a ser possível encontrar semelhanças e diferenças.

### 4.1. Metodologia de Booch

Booch (1996) definiu a notação de que um sistema é analisado a partir de um número de visões, onde cada visão é descrita por um número de modelos e diagramas.

O método descrito por Booch (1996) é constituído, basicamente, pelas fases de **Análise de requisitos**, **Análise do domínio** e **Desenho**. Tem como providenciar uma notação e um processo de desenvolvimento, dando especial atenção ao aspecto da comunicação entre a fase de análise e desenho de um sistema de software Orientado a Objetos. Este método tem um carácter amplo que, endereçando os aspectos das ferramentas de análise e desenho Orientado a Objetos, inclui modelagem dos objetos, modelagem da análise, desenho da aplicação, desenho da implementação e ciclo de vida dos processos.

Booch (1996) propõe diferentes visões para descrever os sistemas. O modelo lógico, isto é, o domínio do problema, é representado na estrutura de classes e objetos. O diagrama de objetos mostra como os objetos interagem uns com os outros, enquanto que os diagramas de classe são de índole mais estática. Assim, os diagramas de objetos descrevem o comportamento dinâmico do sistema. O conceito de subsistema, neste método, é entendido como um diagrama de módulos, tendo um paralelismo, em termos do papel

que representa, com os diagramas de categoria e de classes. Os subsistemas representam conjuntos de módulos relacionados de uma forma lógica.

Segundo Booch (1996), um sistema pode consistir em múltiplos programas, executando num conjunto de computadores distribuídos. O diagrama de processos é utilizado para visualizar a forma como os processos são alocados aos processadores, em termos de desenho físico do sistema. Tipicamente, pode ser incluído apenas um diagrama de processo.

Ainda segundo Booch (1996), a arquitetura de módulos e de processos lida com a alocação física das classes e objetos aos respectivos módulos, processadores, dispositivos, e ligações de comunicação entre eles, isto é, descreve os requisitos de hardware concretos em relação aos componentes de software do sistema.

#### **4.1.1. Conceitos**

De acordo com Booch (1996) este método utiliza uma série de diagramas para descrever as decisões estratégicas tomadas nas fases de análise e desenho, que devem ser consideradas quando é criado um sistema segundo o paradigma dos objetos:

- **Diagramas de classe**
  - **Relacionamentos** – utilizados para indicar ligações semânticas entre as classes
- **Diagramas de transição de estados**
- **Diagramas de Interação de objetos**

#### **4.1.2. Enquadramento Conceitual**

Um dos grandes pressupostos do método de Booch (1996) é criar um modelo do sistema, passível de ser implementado. Os vários conceitos deste método estão distribuídos, com diferentes níveis de abstração, nas três fases do processo, indicados a seguir:

- **Análise de requisitos:** produz dois tipos de especificações formais. A primeira é uma declaração das funções principais do sistema. Devem ser definidos *inputs* e *outputs* do sistema, ou uma lista de referências para planos de ação, procedimentos, etc. A segunda é a especificação de um conjunto de mecanismos chave que o sistema deve providenciar, incluindo o estado de entrada, saída e as mudanças de estado esperadas. Estes mecanismos chave servem para validar o domínio do modelo, descobrir operações, bem como testar determinados casos.
- **Análise do domínio** - esta fase inclui:
  - Diagramas de classe abstratos;
  - Especificações das classes;
  - Herança;
  - Diagramas de cenários dos objetos;
  - Especificação dos objetos.
- **Desenho:** refinamento do modelo criado na fase de análise, adicionando as estruturas e comportamentos necessários para implementação do domínio em estudo. O modelo iniciado durante a análise serve como base para a construção do modelo de desenho. São adicionadas as seguintes contribuições ao modelo:
  - Descrições arquiteturas;
  - Descrições do protótipo;
  - Diagramas de categoria;
  - Diagramas de classe;
  - Diagramas de objeto;
  - Aperfeiçoamentos às especificações das classes.

#### 4.1.3. Processos e técnicas

Para Booch (1996), os utilizadores deste método desenvolvem modelos de um determinado sistema, implementando-os diretamente em unidades de código, denominados protótipos. O modelo é construído em estágios que

permitem a concentração em determinados aspectos do sistema, um determinado momento. Podem ser escolhidos diferentes diagramas para focar áreas críticas do sistema. O resultado é uma série de visões claras e expressivas, como resposta a questões específicas sobre o desenho do sistema ou análise de requisitos. Este método tem como base o desenvolvimento de um sistema como um processo iterativo, isto é, conceitualizações anteriores devem ser complementadas ou refinadas, servindo o resultado como *input* do próximo estágio, de uma forma incremental. Assim, as primitivas codificações da fase de desenho podem servir como forma de descoberta dos requisitos do sistema:

- **Análise de requisitos:** é o processo de determinação daquilo que o cliente quer que o sistema faça. É uma fase de alto nível que identifica as funções chave que o sistema deve desempenhar, definindo o alcance do domínio que o sistema deve suportar, documentando os processos e planos de ação da organização suportados pelo sistema. Os passos desta fase não são definidos formalmente, pois este processo varia de uma forma dramática, já que depende de uma série de fatores.
- **Análise do domínio:** Nesta fase, é definido, de uma forma precisa e concisa, o modelo da organização, relevante para o sistema. É neste processo que, é ganho um detalhe de conhecimento do domínio necessário para que o sistema cumpra as funções exigidas. Durante esta fase, são executados os seguintes passos:
  - Definição das classes;
  - Definição de relações;
  - Procura de atributos;
  - Definição da herança;
  - Definição de operações;
  - Validação e iteração.

Booch (1996) define este processo como sendo altamente iterativo, isto é, podem ser encontrados relacionamentos de herança, quando são descobertos os atributos.

➤ **Desenho:** nesta fase, são determinadas, de forma eficaz, eficiente e de baixo custo, implementações físicas que cumpram a funcionalidade pretendida, bem como o armazenamento dos dados definidos na análise do domínio. É feito o mapeamento da análise lógica, feita durante a fase de análise de domínio, para uma estrutura que permita que os objetos e as classes sejam passíveis de codificação e execução. É feito um refinamento contínuo e estendido ao modelo conseguido durante a fase de análise de domínio através de:

- Determinação da arquitetura inicial;
- Determinação do desenho lógico;
- Mapeamento para a implementação física;
- Refinamento do desenho.

Para Booch (1996) tal como a análise, o desenho também é um processo iterativo e incremental. Pode ser necessário voltar à análise, quando são encontradas ambigüidades e omissões. A iteração também é feita pelos passos desta fase, à medida que os protótipos vão sendo construídos, novas partes do sistema são integradas e protótipos existentes vão sendo estendidos, de forma a construir o sistema completo.

#### 4.1.4. Ferramentas de suporte

Este método é suportado pelas seguintes ferramentas, entre outras:

- ✓ **System Architect**, de New York;
- ✓ **Paradigm Plus**, de Houston;
- ✓ **Palladio Software**, de Wisconsin;
- ✓ **Rational Rose**, da Califórnia;
- ✓ **ObjectMaker**, da Califórnia;
- ✓ **Semaphore Tools**, de Massachusetts.

## 4.2. Metodologia de Rumbaugh

Este método denominado - OMT (*Object Modelling Technique*) – é um método desenvolvido pela GE, onde Rumbaugh trabalhava. Segundo Rumbaugh et al (1994) é voltado para o teste de modelos, baseado nas especificações da análise e requisitos do sistema. Tem um caráter amplo em termos de domínio de alcance, não incluindo a modelagem estratégica. Este método encontra-se dividido em quatro fases:

- **Análise;**
- **Desenho do sistema;**
- **Desenho dos objetos;**
- **Implementação.**

Rumbaugh et al (1994) faz uma grande distinção entre análise e desenho. O modelo de análise é segmentado em três sub-modelos:

- **Modelo de objetos;**
- **Modelo dinâmico;**
- **Modelo funcional.**

Segundo Rumbaugh et al (1994), na fase de desenho, os esforços são concentrados exclusivamente na arquitetura global do sistema, concentrando-se na otimização e refinamento do modelo de objetos, preparando-o para a transformação numa linguagem de programação. É dada pouca importância ao desenvolvimento da interface homem-máquina, embora os diálogos com o utilizador denominados cenários representem uma importante técnica de análise nesta área.

A OMT de Rumbaugh et al (1994) não dá grande atenção ao comportamento do sistema em questão, como forma de descoberta e clarificação das hierarquias de classes existentes. OMT foi aplicado inicialmente com o intuito de desenvolvimento de sistemas de tempo real e sistemas específicos, como compiladores, interfaces gráficas e bases de dados, podendo, no entanto, ser aplicado ao desenvolvimento de sistemas de informação.

### 4.2.1. Conceitos

O método de Rumbaugh et al (1994) suporta os paradigmas básicos dos objetos, como seja: abstração, encapsulamento e herança. O conceito de mensagem não é predominante, sendo o estado, transição e evento largamente explorados no modelo dinâmico. Cobre, no entanto, alguns conceitos menos freqüentemente utilizados por outros métodos, que serão mostrados de seguida:

- ◆ **Modelo de objetos:** em relação aos conceitos associados com o modelo de objetos, destacam-se:
  - **Associação qualificada:** associação binária entre classes;
  - **Subclasse substituta:** subclasses que se sobrepõem nos membros constituintes;
  - **Módulo:** subconjunto coerente de um sistema contendo um grupo de classes relacionadas.
- ◆ **Modelo funcional:** dos conceitos associados com o modelo funcional, destacam-se:
  - **Ator:** objeto ativo que atua sobre o diagrama de fluxo de dados, produzindo ou consumindo dados;
  - **Processo:** algo que transforma os valores dos dados;
  - **Fluxo de dados:** ligação entre o output de um objeto ou processo e o input de outro;
  - **Fluxo de controle:** valor booleano que afeta um processo em execução.
- ◆ **Modelo dinâmico:** dos conceitos associados com o modelo dinâmico, destacam-se:
  - **Ação:** operação instantânea associada com um evento;
  - **Atividade:** operação que demora algum tempo a ser concluída, estando associada com um estado e representando um fato do mundo real.
  - **Condição:** função booleana dos valores de um objeto, válida durante um intervalo de tempo;

- **Condição de guarda:** expressão booleana que deve ser verdadeira de forma a ocorrer uma determinada transição.
- ◆ **Regras:** Em relação aos conceitos associados com as regras, destacam-se:
  - **Restrição:** relação funcional entre objetos, classes, atributos, ligações e associações, que deve ser mantida verdadeira.
  - **Afirmção:** declaração acerca de uma condição ou relacionamento que deve ser verdadeira ou falsa na altura em que é testada;
  - **Invariante:** declaração sobre uma condição ou relacionamento que deve ser sempre verdadeira.

#### 4.2.2. Enquadramento Conceitual

Para Rumbaugh et al (1994) a fase de análise consiste no seguinte:

- **Definição do problema;**
- **Modelo de objetos**, constituído pelos diagrama de objetos e dicionário de dados;
- **Modelo dinâmico**, constituído pelos diagramas de estado e diagramas de fluxo de eventos;
- **Modelo funcional**, constituído pelos diagramas de fluxo de dados e os constrangimentos associados.

Ainda para Rumbaugh et al (1994) a fase de desenho do sistema descreve a arquitetura básica do sistema, bem como decisões estratégicas de alto nível, tratando-se de um refinamento da fase anterior, consistindo em:

- **Modelo de objetos detalhado;**
- **Modelo dinâmico detalhado;**
- **Modelo funcional detalhado.**

#### 4.2.3. Processos

Segundo Rumbaugh et al (1994) algumas atividades do método podem ser feitas em paralelo, se for necessário, e, depois da análise, os subsistemas podem ser desenhados e implementados independentemente. Mas, existe uma grande distinção entre análise e desenho, o que não é tão notório noutros métodos Orientado a Objetos. Os processos neste método são descritos por Rumbaugh et al (1994) em quatro fases distintas, que são apresentados a seguir:

➤ **Análise**

- Escrever ou obter a definição do problema;
- Construir o modelo de objetos, com as suas sub-atividades;
- Desenvolver o modelo dinâmico, com as suas sub-atividades;
- Construir o modelo funcional, com as suas sub-atividades;
- Verificar, iterar, e refinar os três modelos, com as sub-atividades.

➤ **Desenho do sistema**

- Organizar o sistema em subsistemas;
- Identificar a concorrência inerente ao problema;
- Alocar os subsistemas aos processadores e tarefas;
- Encontrar uma estratégia básica para implementar o armazenamento dos dados;
- Determinar mecanismos para controlar o acesso aos recursos globais;
- Escolher a natureza da implementação;
- Manipular as condições de fronteira.

➤ **Desenho dos objetos**

- Obter operações do modelo funcional e dinâmico;
- Desenhar os algoritmos que implementam as operações;
- Otimizar os caminhos de acesso aos dados;
- Ajustar a estrutura das classes de forma a permitir a herança entre as mesmas;
- Desenhar a implementação das associações;
- Determinar a representação exata dos atributos dos objetos;
- Colocar as classes e associações em módulos.

➤ **Implementação**

- Desenho da base de dados;
- Escrita do código.

#### 4.2.4. Técnicas

➤ **Análise**

- Notação para diagramas de modelos de objetos com caráter amplo;
- Cenários utilizados para construir modelos dinâmicos;
- Notação para diagramas de estado, podendo as ações estar ligadas às transições e aos estados;
- Diagramas de fluxos de dados utilizados na modelagem funcional;
- Técnicas para encontrar operações a partir dos eventos, funções etc.;
- Iteração.

➤ **Desenho do sistema**

- Decomposição do sistema em níveis ou divisões;
- Utilização de protótipos arquiteturais.

➤ **Desenho de objetos**

- Otimização, introduzindo redundância;
- Aproximações para implementação dos modelos de estados;
- Técnicas para aumentar a reutilização, incluindo delegação e exploração da herança;
- Técnicas de desenho das associações;
- Técnicas de representação dos atributos.

➤ **Implementação**

- Diretrizes de estilo de programação, de forma a aumentar a reutilização, extensibilidade, robustez e entendimento;
- Mapeamento do desenho Orientado a Objetos para a linguagem Orientada a Objetos;

- Mapeamento do desenho Orientado a Objetos para uma linguagem não Orientada a Objetos;
- Mapeamento do modelo de objetos para o desenho de uma base de dados relacional.

#### 4.2.5. Ferramentas de suporte

A notação **OMT** (*Object Modeling Technique*) é suportado por algumas ferramentas CASE existentes, dentre elas **o Paradigm Plus** e o **OMTool**.

#### 4.3. Metodologia de Jacobson

A metodologia de Jacobson et al (1992) – *Object Oriented Software Engineering (OOSE)* – é baseada na utilização de “*use cases*”, que definem os requisitos iniciais do sistema, vistos por um ator externo. Este método é adequado para o desenvolvimento de software numa escala industrial.

De acordo com Jacobson et al (1992) o método divide o desenvolvimento em processos, que, diferentemente das fases de desenvolvimento tradicionais, podem ser iterados e sobrepostos. Estes processos produzem uma série de modelos interligados, facilitando posteriormente a junção dos mesmos através de uma modelagem consistente.

Para Jacobson et al (1992) todos os sistemas se alteram durante o seu ciclo de vida, tendo a manutenção dos mesmos um peso muito grande em termos de custos de desenvolvimento. Muitos métodos de desenvolvimento adequam-se a novos desenvolvimentos, mas tratando as revisões do modelo de uma forma pouco adequada. Os desenvolvimentos iniciais devem ser vistos como uma atividade importante, estabelecendo uma arquitetura e uma filosofia que constitui a base do sistema.

Jacobson et al (1992) descreve a metodologia como um conjunto de processos que interagem entre si durante o desenvolvimento do projeto, através de diferentes modelos. Os processos principais são a **análise**, **construção** e **teste**. No processo de análise é criado um modelo conceitual do sistema a construir. Nesta fase, os modelos são desenvolvidos de forma a

facilitar a compreensão do sistema, privilegiando a comunicação com o cliente. No processo de construção, é desenvolvido o sistema a partir dos modelos criados anteriormente. A construção inclui o desenho e a implementação, resultando um sistema completo. O processo de teste integra os componentes do sistema, testando-o e decidindo se está pronto a ser distribuído ao cliente.

Segundo Jacobson et al (1992) o conhecimento do sistema aumenta sucessivamente à maneira que o trabalho vai progredindo. Quando se está trabalhando na primeira versão de desenvolvimento do sistema, surgem novos requisitos e outros são alterados. Deste modo o sistema não pode ser completamente desenvolvido, até que as especificações dos requisitos se mantenham constantes. Uma forma de resolver este problema é através de desenvolvimento incremental. Na prática, o sistema é dividido em partes, correspondentes aos serviços requeridos pelo cliente. Cada novo estágio de desenvolvimento estende o sistema com nova funcionalidade até que o produto esteja terminado. De tal modo que, esta estratégia incremental providencia um grande *feedback* durante o processo de desenvolvimento, resultando na execução dos processos várias vezes durante a mesma versão do sistema.

Ainda segundo Jacobson et al (1992) nesta aproximação, o modelo de casos de uso (*use cases*) é o modelo principal no qual todos os outros modelos são derivados. Um modelo de *use cases* descreve a funcionalidade completa do sistema, identificando como é que o sistema externo interage com o sistema. O modelo de *use cases* é a base das fases de análise, construção e teste.

O objetivo da análise é compreender o sistema de acordo com os seus requisitos funcionais. Os objetos são encontrados, organizados e as suas interações são descritas. As operações dos objetos e a visão interna é descrita durante a fase de análise. É importante que os objetos sejam encontrados na fase de análise. Na fase de teste, o sistema é verificado, significando que a correção do sistema é verificada de acordo com a sua especificação (JACOBSON et al, 1992).

#### **4.3.1. Conceitos**

Para Jacobson et al (1992) o primeiro modelo a ser desenvolvido é a especificação dos requisitos. Consiste num modelo orientado ao caso em estudo, com descrições da interface com o utilizador, e um modelo dos objetos do domínio. Este modelo é baseado nos conceitos de *actors* (atores) e *use cases* (casos de uso). Deste modo, define-se o que existe fora do sistema e o que deve ser executado pelo sistema, respectivamente. Os *actors* podem ser pessoas ou máquinas. Cada um deles pode executar um número diferente de tarefas, bem como participar na operação do sistema. Executa uma seqüência de transações em permanente diálogo com o sistema. A uma seqüência específica é denominado *use case*. Cada *use case* é uma forma específica de utilizar o sistema. O conjunto de *use cases* corresponde aos requisitos funcionais do sistema a construir.

Ainda, para Jacobson et al (1992), o modelo de requisitos facilita a discussão sobre os mesmos e as preferências do sistema, de forma a assegurar a definição correta do mesmo. Para suportar o modelo de use cases é apropriado o desenvolvimento de modelos de interface, tendo os protótipos um papel importante na sua simulação. Adicionalmente, para comunicar com os usuários potenciais e obter uma concordância estável das descrições dos *use cases*, um modelo lógico do domínio dos objetos se torna apropriado.

Segundo Jacobson et al (1992) o modelo de requisitos formula as especificações dos requisitos funcionais baseadas nas necessidades dos usuários do sistema.

Ainda, segundo Jacobson et al (1992), uma vez definido e aprovado o modelo de requisitos, o desenvolvimento do sistema passa para a construção do modelo. Este modelo define a estrutura lógica do sistema, independentemente do ambiente de implementação selecionado. Embora possa ser utilizado o modelo de objetos anteriormente definido na fase de análise, como base para a implementação, isto não resulta numa estrutura robusta. O modelo de análise representa uma estrutura mais estável e fácil manutenção.

De acordo com Jacobson et al (1992), o modelo reconhece três tipos diferentes de objetos, separando o controle e coordenação, da funcionalidade relacionada com a interface e das entidades que personificam os conceitos da

área de aplicação. Desta forma, os tipos de objetos utilizados são **entidade**, **interface** e **controle**.

Para Jacobson et al (1992), cada tipo de objeto tem um objetivo diferente. Os objetos entidade modelam a informação do sistema que deve ser utilizada por um determinado período de tempo. Todo o comportamento ligado a esse tipo de informação deve ser colocado neste tipo de objetos. Os objetos interface modelam o comportamento e informação que depende do atual interface do sistema. Os objetos controle modelam a funcionalidade que não é genérica, mas específica a um ou mais *use cases*, comportamento que consiste em operações sobre diferentes objetos entidade, executando algumas computações e retornando o resultado para um objeto interface.

Este modelo segundo Jacobson et al (1992) não obriga à criação de um modelo rígido baseado nestes tipos de objetos. O resultado é um modelo robusto de uma aplicação, isto é, um sistema que é, fundamentalmente, mais fácil de ser adaptado a extensões e alterações, junto com conjuntos de componentes que são mais facilmente reutilizáveis. Assim, futuras alterações ao sistema, não afetam a estrutura lógica do mesmo. Este modelo apresenta os seguintes conceitos:

- **Actor** (ator): define um papel que um determinado utilizador pode representar na sua troca de informação com o sistema;
- **Role** (papel): o papel é definido pelas operações dos objetos;
- **User** (usuário): pessoa que atualmente utiliza o sistema. É uma instância da classe *actor*;
- **Use Case** (caso de uso): curso completo de eventos especificando as ações entre o utilizador e o sistema;
- **Object** (objeto): é caracterizado pelas operações e estado associado;
- **Entity Object** (objeto entidade): informação que é guardada por um largo período de tempo, mesmo quando uma use case é completada;
- **Interface Object** (objeto interface): objeto que contém funcionalidade da *use case* que interage diretamente com o ambiente;
- **Control Object** (objeto controle): objeto que modela a funcionalidade que não está em qualquer outro objeto;

- **Central Interface Object** (objeto interface central): objeto de interface que contém outros objetos de interface;
- **Attribute** (atributo): contém informação e o seu tipo;
- **Class** (classe): grupo de objetos que têm comportamento e estruturas de informação similares;
- **Stimulus** (estímulo): um evento que é enviado de um objeto para outro, iniciando uma determinada operação;
- **Message** (mensagem): estímulo intra-processo, isto é, uma chamada normal dentro de um processo;
- **Signal** (sinal): estímulo intra-processo, mas uma chamada dentro de dois processos;
- **Operation** (operação): atividade dentro de um bloco que pode levar a uma mudança de estado do objeto correspondente;
- **Subsystem** (subsistema): grupo definido de objetos, de forma a estruturar o sistema.

#### 4.3.2. Processos

Jacobson et al (1992), define que, durante o desenvolvimento, o método é dividido em modelos de sistema. Os modelos desenvolvidos descrevem o que é que o sistema faz. O desenvolvimento desses modelos depende da descrição dos processos que fazem parte do sistema:

- ✓ **Modelo de requisitos:** a base deste modelo são os requisitos do cliente ou usuário final, expressos de uma determinada forma. É a compreensão por parte do analista, daquilo que o cliente quer, que deve ser expressa, podendo servir como um contrato entre o analista e o cliente. Este modelo consiste, normalmente, no modelo de *use cases* com as descrições de interface e o modelo dos objetos do domínio em questão. O sistema é descrito como um número de *use cases* que são executados pelos *actors*. Os *actors* constituem aquilo que é externo ao sistema a ser desenvolvido, podendo os limites do sistema serem estabelecidos. Cada um deles executa uma ou mais tarefas. Uma vez definido aquilo que está fora do sistema, a

funcionalidade interna pode ser definida através da especificação de *use cases*. Os *actors* são a principal ferramenta para encontrar *use cases*. Um *use case* é uma forma específica de utilizar alguma da funcionalidade total do sistema. Como os *use cases* focam certas áreas funcionais da aplicação, é possível fazer o seu desenvolvimento para diferentes áreas, juntando-as mais tarde para completar o modelo de requisitos do sistema;

- ✓ **Modelo de análise:** o objetivo deste modelo é encontrar a consistência necessária e a estrutura do sistema como uma base para a construção. Cada tipo de objeto tem o seu objetivo especial para a robustez, e juntos, oferecem a funcionalidade total especificada no modelo de requisitos. Toda funcionalidade que está diretamente dependente do ambiente do sistema é colocado nos objetos de interface, sendo através destes objetos que os *actors* se comunicam com o sistema. A tarefa dos objetos de interface é traduzir as ações dos *actors* em eventos do sistema. Para modelar a informação que o sistema deve manipular ao longo do tempo são utilizados os objetos entidade. Quando o comportamento não pode ser colocado de uma forma natural nestes dois tipos de objetos, são colocados nos objetos do tipo controle. Os objetos identificados são agrupados em pacotes, servindo como *input* para a fase de desenho. Estes pacotes reduzem a complexidade e estrutura do sistema, de forma a facilitar posteriores desenvolvimentos e manutenção do próprio sistema, e todos os objetos relacionados funcionalmente devem ser colocados no mesmo pacote.
- ✓ **Modelo de desenho:** o modelo de desenho é um refinamento e formalização do modelo de análise, onde as consequências do ambiente da implementação têm de ser tidas em conta. O modelo de análise não é suficientemente formal, e, em vez de estarmos permanentemente a alterar o código fonte, procedesse ao refinamento dos objetos, às operações que eles devem oferecer, às comunicações exatas que devem ter entre eles e que estímulos é que são enviados. O modelo define explicitamente as interfaces dos objetos, bem como as semânticas das operações, e reflete decisões

relacionadas com as bases de dados, linguagens de programação e distribuição. É composto por pacotes e design *objects*, onde depois de identificar a arquitetura do sistema, deve ser descrito como é que os pacotes e os design *objects* se comunicam. Para cada *use case* concreto deve ser desenhado um diagrama de interação, descrevendo como é que o *use case* é realizado através da comunicação entre os pacotes, que recebem e enviam estímulos. Estes estímulos, incluindo os parâmetros enviados, devem ser definidos. Os diagramas de transição de estado podem ser utilizados num nível intermédio, antes de iniciar a implementação. Estes diagramas descrevem que estímulos podem ser recebidos e o que acontece quando o estímulo é recebido.

- ✓ **Modelo de implementação:** a implementação dos pacotes em código fonte pode iniciar-se quando a interface do pacote estabilizar. Em muitos casos, cada *design object* corresponde exatamente a uma classe, tornando fácil o mapeamento. Em alguns casos, classes podem ser implementadas para um dado *design object*, para implementação dos atributos, utilização de componentes ou separação de funcionalidade comum em classes abstratas. Por conseguinte, cada *design object* continuará a ter um mapeamento no código, mas suportando várias classes que foram adicionadas. Se possível, duas classes não devem oferecer funcionalidade semelhante, a não ser que estejam relacionadas por um mecanismo de herança. É evidente que leva tempo para desenhar boas classes, e nem sempre é eficiente desenvolver todas as classes do sistema o mais genéricas possível. Note-se que é mantida uma linha de ação desde a fase de análise até ao código fonte. Quando se lê o código fonte, pode-se descobrir diretamente o que é que o originou no modelo de requisitos.
- ✓ **Modelo de teste:** nesta fase, é feita a verificação do sistema construído em termos de correção. Uma aproximação disciplinada e bem organizada ao desenvolvimento de sistemas resulta da necessidade de aumentar a qualidade do sistema, diminuindo os custos da fase de teste. Na metodologia, as atividades de teste são

executadas inteiramente durante o processo de desenvolvimento. As instâncias de diferentes classes devem ser integradas continuamente ao longo do processo. O conceito de *use case* é crucial para a integração, com um *use case* em cada momento a ser integrado. Os planos de teste podem ser feitos muito cedo, juntos com a identificação e especificação dos *use cases*.

### **4.3.3. Técnicas**

Uma das técnicas utilizada por Jacobson et al (1992) são os diagramas de interação, que descrevem de que maneira é que cada *use case* funciona, através da interação dos diversos objetos.

De acordo com Jacobson et al (1992), também é representada a interface com tudo aquilo que está fora do diagrama, como sejam os *actors* externos, que podem corresponder a interfaces externas do sistema. A descrição das seqüências é feita de uma forma textual, tendo estas construções a facilidade de migração para a linguagem de programação a ser adotada.

Além desta técnica, são utilizadas outras definidas em modelos de requisitos, análise, desenho, implementação e testes. Utiliza também outras técnicas divulgadas noutros métodos de desenvolvimento orientados a objetos.

### **4.3.4. Ferramentas de suporte**

Este método disponibiliza várias configurações, cada uma delas para casos específicos de desenvolvimento. Assim, o processo e as ferramentas são utilizados de acordo com os casos específicos de desenvolvimento, por exemplo para que um objeto tenha o mesmo nome e características na documentação, uma ferramenta que suporte links de hipertexto, descrita por Bergner (1990), simplifica e dá consistência à tarefa.

#### 4.4. Metodologia UML

A UML (*Unified Modeling Language*) é uma tentativa de padronizar a modelagem orientada a objetos de uma forma que qualquer sistema, seja qual for o tipo, possa ser modelado corretamente, com consistência, fácil de se comunicar com outras aplicações, simples de ser atualizado e compreensível.

Existem várias metodologias de modelagem orientada a objetos que até o surgimento da UML causavam muitas discussões entre a comunidade de desenvolvedores orientado a objetos. O surgimento da UML pôs fim a esta fase polêmica, trazendo as melhores idéias de cada uma destas metodologias. É uma linguagem de modelagem unificada, que busca uma padronização da linguagem para visualização, especificação, construção e documentação de sistemas, estabelecida pela OMG (*Object Management Group*).

Booch, Rumbaugh e Jacobson (2000) definem a UML como uma linguagem de modelagem e não uma metodologia, ou seja, não tem noções do processo. Um método consiste de uma linguagem de modelagem e de um processo. A linguagem de modelagem é a notação gráfica utilizada por métodos para desenhar projetos. O processo é o caminho de quais passos serão utilizados na elaboração de um projeto, ou seja, pode-se utilizar os conceitos de UML, sem ficar preso a um processo padrão, permitindo que o desenvolvedor faça o seu próprio processo de acordo com as necessidades, em função do tipo de *software* a ser desenvolvido (tempo real, sistema de informações, produto *desktop*), o tamanho da equipe a ser envolvida, o nível de formalismo exigido e a facilidade de comunicação desejada, conferindo assim uma flexibilidade a ser usada com todos os tipos de processos e em todo o ciclo de desenvolvimento de software.

##### 4.4.1. Objetivos

Para Booch, Rumbaugh e Jacobson (2000) os objetivos da UML são:

- A modelagem de sistemas (não apenas de software) usando os conceitos da orientação a objetos;

- Estabelecer uma união fazendo com que métodos conceituais sejam também executáveis;
- Criar uma linguagem de modelagem usável tanto pelo homem quanto pela máquina.

O paradigma de Orientação a Objetos utilizando a UML descrita por Booch, Rumbaugh e Jacobson (2000), introduz uma nova abordagem no ciclo de desenvolvimento de sistemas, adotando um processo de produção de software de forma iterativa e incremental, ou seja, passando diversas vezes pelas fases previstas no processo tradicional em cascata, agregando em cada “passagem” novas funcionalidades e correções observadas nas versões anteriores.

#### 4.4.2. Ciclo de Desenvolvimento

Segundo Booch, Rumbaugh e Jacobson (2000), as fases no desenvolvimento de sistemas de software utilizando UML são: análise de requisitos, análise, design (projeto), programação e testes. Estas fases não precisam ser executadas na ordem descrita acima, mas concomitantemente de forma que problemas detectados numa certa fase modifiquem e melhorem as fases desenvolvidas anteriormente de forma que o resultado global gere um produto de alta qualidade e performance. A seguir Booch, Rumbaugh e Jacobson (2000) descrevem como cada uma as fases do ciclo de desenvolvimento de sistemas estão inseridas dentro do contexto da Orientação a Objetos utilizando a UML:

- ❖ **Análise de Requisitos:** esta fase captura as intenções e necessidades dos usuários do sistema a ser desenvolvido através do uso de funções chamadas *use cases*. Através do desenvolvimento de *use cases*, as entidades externas ao sistema (em UML chamados de "atores externos") que interagem e possuem interesse no sistema são modelados entre as funções que eles requerem, funções estas chamadas de *use cases*. Os atores externos e os *use cases* são modelados com relacionamentos que possuem comunicação associativa entre eles ou são desmembrados em hierarquia. Cada

*use cases* modelado é descrito através de um texto, e este especifica os requerimentos do ator externo que utilizará este *use cases*. O diagrama de *use cases* mostrará o que os atores externos, ou seja, os usuários do futuro sistema deverão esperar do aplicativo, conhecendo toda sua funcionalidade sem importar como esta será implementada. A análise de requisitos também pode ser desenvolvida baseada em processos de negócios, e não apenas para sistemas de software;

- ❖ **Análise:** a fase de análise está preocupada com as primeiras abstrações (classes e objetos) e mecanismos que estarão presentes no domínio do problema. As classes são modeladas e ligadas através de relacionamentos com outras classes, e são descritas no Diagrama de Classe. As colaborações entre classes também são mostradas neste diagrama para desenvolver os *use cases* modelados anteriormente, estas colaborações são criadas através de modelos dinâmicos em UML. Na análise, só serão modeladas classes que pertençam ao domínio principal do problema do software, ou seja, classes técnicas que gerenciem banco de dados, interface, comunicação, concorrência e outros não estarão presentes neste diagrama;
- ❖ **Design (Projeto):** na fase de *design*, o resultado da análise é expandido em soluções técnicas. Novas classes serão adicionadas para prover uma infra-estrutura técnica: a interface do usuário e de periféricos, gerenciamento de banco de dados, comunicação com outros sistemas, dentre outros. As classes do domínio do problema modeladas na fase de análise são mescladas nessa nova infra-estrutura técnica tornando possível alterar tanto o domínio do problema quanto à infra-estrutura. O *design* resulta no detalhamento das especificações para a fase de programação do sistema;
- ❖ **Programação:** na fase de programação, as classes provenientes do *design* são convertidas para o código da linguagem orientada a objetos escolhida (a utilização de linguagens procedurais não é recomendada). Dependendo da capacidade da linguagem utilizada, essa conversão pode ser uma tarefa fácil ou muito complicada. No

momento da criação de modelos de análise e *design* em UML, é melhor evitar traduzi-los mentalmente em código. Nas fases anteriores, os modelos criados são o significado do entendimento e da estrutura do sistema, então, no momento da geração do código onde o analista conclua antecipadamente sobre modificações em seu conteúdo, seus modelos não estarão mais demonstrando o real perfil do sistema. A programação é uma fase separada e distinta, onde os modelos criados são convertidos em código;

- ❖ **Testes:** um sistema normalmente é rodado em testes de unidade, integração, e aceitação. Os testes de unidade são para classes individuais ou grupos de classes e são geralmente testados pelo programador. Os testes de integração são aplicados já usando as classes e componentes integrados para se confirmar se as classes estão cooperando uma com as outras como especificado nos modelos. Os testes de aceitação observam o sistema como uma "caixa preta" e verificam se o sistema está funcionando como o especificado nos primeiros diagramas de *use cases*. O sistema será testado pelo usuário final e verificará se os resultados mostrados estão realmente de acordo com as intenções do usuário final.

#### 4.4.3. Notação da UML

Na UML Booch, Rumbaugh e Jacobson (2000) definem alguns elementos cujas notações podem ser empregadas ao longo dos diversos diagramas propostos. Os principais são:

- **Pacote:** é um mecanismo de propósito geral, para organizar elementos do modelo em grupos, podendo estar dentro de outros pacotes (subordinados). Ele é utilizado para permitir uma melhor visualização de sistemas muito complexos, reunindo os elementos que tenham uma coesão lógica funcional, resultando em porções principais que são os pacotes. Por exemplo, se grupos de classes relacionam-se entre si através de uma junção forte ou ainda se for possível dar um nome a um grupo de classes, isso pode indicar a

existência de um bom agrupamento para pacote. São úteis não só para designar agrupamentos lógicos e físicos, mas também para designar grupos de *use cases* e grupos de processador.

- **Estereótipo:** permite a introdução de novos elementos para se estender a capacidade de modelagem da linguagem. Permite estender a semântica, mas não a estrutura preexistente ou classes, além de prover um grau de liberdade aos usuários para ajustar a linguagem às suas necessidades. Um estereótipo para classe é escrito textualmente ou como um ícone no canto direito superior.
- **Nota:** é um comentário colocado em um diagrama sem qualquer conteúdo semântico.

#### 4.4.4. Diagramas

Booch, Rumbaugh e Jacobson (2000) descrevem os diagramas utilizados pela UML a seguir:

- **Diagrama de *use-case*:** a modelagem de um diagrama *use case* é uma técnica usada para descrever e definir os requisitos funcionais de um sistema. Eles são escritos em termos de atores externos, *use cases* e o sistema modelado. Os atores representam o papel de uma entidade externa ao sistema como um usuário, um hardware, ou outro sistema que interage com o sistema modelado. Os atores iniciam a comunicação com o sistema através dos *use cases*, onde o *use case* representa uma seqüência de ações executadas pelo sistema e recebe do ator que lhe utiliza dados tangíveis de um tipo ou formato já conhecido, e o valor de resposta da execução de um *use case* (conteúdo) também já é de um tipo conhecido, tudo isso é definido juntamente com o *use case* através de texto de documentação. O uso de *use cases* em colaborações é muito importante, onde estas são a descrição de um contexto mostrando classes/objetos, seus relacionamentos e sua interação exemplificando como as classes/objetos interagem para executar

uma atividade específica no sistema. Uma colaboração é descrita por diagramas de atividades e um diagrama de colaboração;

- **Diagrama de classes:** o diagrama de classes demonstra a estrutura estática das classes de um sistema onde estas representam as "partes" que são gerenciadas pela aplicação modelada. Classes podem se relacionar com outras através de diversas maneiras: associação (conectadas entre si), dependência (uma classe depende ou usa outra classe), especialização (uma classe é uma especialização de outra classe), ou em pacotes (classes agrupadas por características similares). Todos estes relacionamentos são mostrados no diagrama de classes juntamente com as suas estruturas internas, que são os atributos e operações. O diagrama de classes é considerado estático já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema. Um sistema normalmente possui alguns diagramas de classes, já que não são todas as classes que estão inseridas em um único diagrama e uma certa classe pode participar de vários diagramas de classes;
- **Diagrama de objetos:** o diagrama de objetos é uma variação do diagrama de classes e utiliza quase a mesma notação. A diferença é que o diagrama de objetos mostra os objetos que foram instanciados das classes. O diagrama de objetos é como se fosse o perfil do sistema em um certo momento de sua execução. A mesma notação do diagrama de classes é utilizada com duas exceções: os objetos são escritos com seus nomes sublinhados e todas as instâncias num relacionamento são mostradas. Os diagramas de objetos não são tão importantes como os diagramas de classes, mas eles são muito úteis para exemplificar diagramas complexos de classes ajudando muito em sua compreensão;
- **Diagrama de estado:** o diagrama de estado é tipicamente um complemento para a descrição das classes. Este diagrama mostra todos os estados possíveis que objetos de uma certa classe podem se encontrar e mostram também quais são os eventos do sistema que provocam tais mudanças. Não são escritos para todas as classes de um sistema, mas apenas para aquelas que possuem um

número definido de estados conhecidos e onde o comportamento das classes é afetado e modificado pelos diferentes estados. Capturam o ciclo de vida dos objetos, subsistemas e sistemas. Eles mostram os estados que um objeto pode possuir e como os eventos (mensagens recebidas, timer, erros, e condições sendo satisfeitas) afetam estes estados com o passar do tempo. Diagramas de estado possuem um ponto de início e vários pontos de finalização. Um ponto de início (estado inicial) é mostrado como um círculo todo preenchido, e um ponto de finalização (estado final) é mostrado como um círculo em volta de um outro círculo menor preenchido. Um estado é mostrado como um retângulo com cantos arredondados. Entre os estados estão as transições, mostrados como uma linha com uma seta no final de um dos estados. A transição pode ser nomeada com o seu evento causador;

- **Diagrama de seqüência:** um diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos de um sistema. O mais importante aspecto deste diagrama é que a partir dele percebe-se a seqüência de mensagens enviadas entre os objetos. Ele mostra a interação entre os objetos, alguma coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de seqüência consiste em um número de objetos mostrados em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam. Os diagramas de seqüência podem mostrar objetos que são criados ou destruídos como parte do cenário documentado pelo diagrama. Um objeto pode criar outros objetos através de mensagens. A mensagem que cria ou destrói um objeto é geralmente síncrona, representada por uma seta sólida;
- **Diagrama de Colaboração:** um diagrama de colaboração mostra de maneira semelhante ao diagrama de seqüência, a colaboração dinâmica entre os objetos. Normalmente pode-se escolher entre utilizar o diagrama de colaboração ou o diagrama de seqüência. No diagrama de colaboração, além de mostrar a troca de mensagens

entre os objetos, percebem-se também os objetos com os seus relacionamentos. A interação de mensagens é mostrada em ambos os diagramas. Se a ênfase do diagrama for o decorrer do tempo, é melhor escolher o diagrama de seqüência, mas se a ênfase for o contexto do sistema, é melhor dar prioridade ao diagrama de colaboração. É desenhado como um diagrama de objeto, onde os diversos objetos são mostrados juntamente com seus relacionamentos. As setas de mensagens são desenhadas entre os objetos para mostrar o fluxo de mensagens entre eles. As mensagens são nomeadas, que entre outras coisas mostram a ordem em que as mensagens são enviadas;

- **Diagrama de atividade:** diagramas de atividade capturam ações e seus resultados. Eles focam o trabalho executado na implementação de uma operação (método), e suas atividades numa instância de um objeto. Os estados no diagrama de atividade mudam para um próximo estágio quando uma ação é executada (sem ser necessário especificar nenhum evento como no diagrama de estado). Um diagrama de atividade é uma maneira alternativa de se mostrar interações, com a possibilidade de expressar como as ações são executadas, o que elas fazem (mudanças dos estados dos objetos), quando elas são executadas (seqüência das ações), e onde elas acontecem (*swimlanes*). Um diagrama de atividade pode ser usado com diferentes propósitos inclusive para:
  - capturar os trabalhos que serão executados quando uma operação é disparada (ações). Este é o uso mais comum para o diagrama de atividade;
  - capturar o trabalho interno em um objeto;
  - mostrar como um grupo de ações relacionadas pode ser executado, e como elas vão afetar os objetos em torno delas;
  - mostrar como uma instância pode ser executada em termos de ações e objetos;
  - mostrar como um negócio funciona em termos de trabalhadores (atores), fluxos de trabalho, organização, e objetos (fatores físicos e intelectuais usados no negócio).

- **Diagrama de componente:** o diagrama de componente e o de execução são diagramas que mostram o sistema por um lado funcional, expondo as relações entre seus componentes e a organização de seus módulos durante sua execução. O diagrama de componente descreve os componentes de software e suas dependências entre si, representando a estrutura do código gerado. Os componentes são a implementação na arquitetura física dos conceitos e da funcionalidade definidos na arquitetura lógica (classes, objetos e seus relacionamentos). Eles são tipicamente os arquivos implementados no ambiente de desenvolvimento. Componentes são tipos, mas apenas componentes executáveis podem ter instâncias. Um diagrama de componente mostra apenas componentes como tipos. Para mostrar instâncias de componentes, deve ser usado um diagrama de execução, onde as instâncias executáveis são alocadas em nós (*nodes*). Componentes podem definir interfaces que são visíveis para outros componentes. As interfaces podem ser tanto definidas ao nível de codificação (como em Java) quanto em interfaces binárias usadas em *run-time* (como em OLE). Dependências entre componentes podem então apontar para a interface do componente que está sendo usada;
- **Diagrama de Execução:** o diagrama de execução mostra a arquitetura física do hardware e do software no sistema. Pode mostrar os atuais computadores e periféricos, juntamente com as conexões que eles estabelecem entre si e pode mostrar também os tipos de conexões entre esses computadores e periféricos. Especificam-se também os componentes executáveis e objetos que são alocados para mostrar quais unidades de software são executados e em que destes computadores são executados. Demonstra a arquitetura *run-time* de processadores, componentes físicos (*devices*), e de software que rodam no ambiente onde o sistema desenvolvido será utilizado. É a última descrição física da topologia do sistema, descrevendo a estrutura de hardware e software que executam em cada unidade. É composto por

componentes, que possuem a mesma simbologia dos componentes do diagrama de componentes, nós (*nodes*), que significam objetos físicos que fazem parte do sistema, podendo ser uma máquina cliente numa LAN, uma máquina servidora, uma impressora, um roteador, etc., e conexões entre estes nós e componentes que juntos compõem toda a arquitetura física do sistema.

Para Booch, Rumbaugh e Jacobson (2000) todos os sistemas possuem uma estrutura estática e um comportamento dinâmico. A UML suporta:

- **Modelos estáticos** (estrutura estática), classes e relacionamentos:
  - ✓ Diagrama de classes
  - ✓ Diagrama de objetos
- **Modelos dinâmicos** (comportamento dinâmico):
  - ✓ Diagrama de estado
  - ✓ Diagrama de seqüência
  - ✓ Diagrama de colaboração
  - ✓ Diagrama de atividade
- **Modelo funcional:**
  - ✓ Diagrama de componente
  - ✓ Diagrama de execução

## 5. Bancos de Dados e Linguagens Orientados a Objeto

### 5.1. Bancos de Dados Orientados a Objeto

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 orientada 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 escolhida, como um banco de dados puramente relacional.

Segundo Golendziner (1995) um BDOO deve satisfazer, no mínimo, aos seguintes requisitos:

- **Aplicações:**
  - Manter a história de um objeto (ordenamento)
  - Manter a confiabilidade dos dados em ambientes com concorrência
  - Tratar mudanças no esquema do banco de dados
  - Suporte ao trabalho cooperativo
- **Versões:**
  - Descrição de um objeto em um determinado momento, ou sob um certo ponto de vista
  - Objeto de primeira classe
  - Pode ser manipulada e consultada
- **Objeto versionado:**
  - Objeto de primeira classe
  - Agrupa versões de uma mesma entidade do mundo real
  - Conceito novo em relação aos demais conceitos do paradigma de orientação a objetos

- Objetos (versionado ou não) com as mesmas propriedades e comportamento são agrupados em classes
- **Versão corrente:**
  - Mantida automaticamente pelo sistema
  - Última versão criada
  - Usuário pode especificar versão diferente
- **Relacionamento entre versões:**
  - Representam a evolução do objeto no tempo
- **Propriedades das versões:**
  - Reflete o estágio de desenvolvimento e/ou consistência da versão
- **Objeto Genérico:**
  - Representa um objeto do mundo real
  - Representa a abstração das versões de um objeto de projeto
  - Possui identificação própria
- **Mecanismos de herança:**
  - Entidade modelada como um [único objeto, que é instância da classe mais especializada
  - Cada objeto em uma subclasse possui um ascendente na superclasse
- **Configuração:**
  - Ligação entre o objeto composto e cada um dos seus objetos componentes
  - Uma versão configurada é sempre criada a partir de uma versão existente
  - Possui identificação única
  - Possui ascendentes e descendentes nos vários níveis da hierarquia de herança
  - Está associada a um objeto versionado
- **Extensões futuras:**
  - Especificar uma linguagem para definição de configurações
  - Estender o conceito de objeto versionado para classe versionada
  - Investigar como o modelo responde a outras classes de aplicações
  - Implementar o modelo em outros sistemas orientados o objetos

Segundo Rumbaugh et al (1994), em geral, existem dois tipos de consultas em bancos de dados: as baseadas em conjuntos e as de navegação. Os BDs relacionais são projetados para executar operações paralelas em grandes conjuntos de dados. De modo inverso, as linguagens de programação baseada em objetos são eficientes na navegação rápida de um objeto para outro através da travessia de ponteiros. Um BD relacional executa a navegação utilizando junções, que são mais lentas do que a travessia de ponteiros. Um BDOO deve executar eficientemente ambos os tipos de consulta. Uma característica importante de um BDOO é a pressuposição implícita de que o sistema é baseado em operações com objetos isolados.

BDOOs devem ser utilizados como repositório final de dados em uma metodologia de desenvolvimento que utiliza a Análise Orientada a Objeto (AOO), o Projeto Orientado a Objeto (PrOO) e a Programação Orientada a Objeto (POO). Embora existam BDOOs comercialmente disponíveis, as opções são poucas e a tecnologia não está totalmente sedimentada, e por este motivo, não são utilizados em grande escala. 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 parte dos benefícios do paradigma.

## **5.2. Linguagens Orientadas a Objeto**

Para Rumbaugh et al (1994), não é de surpreender que as linguagens baseadas em objetos sejam a meta de implementação mais natural para um projeto baseado em objetos. As linguagens Orientadas a Objeto (LOO) representam um paradigma em relação as linguagens procedurais. Mesmo quando suportadas por algumas extensões de linguagens tradicionais como o Pascal e o C, elas contém características próprias como os objetos, classes, mensagens, métodos e hereditariedade.

Podemos conceituar uma linguagem orientada a objetos, através da análise de uma linguagem puramente orientada a objeto, a *Smalltalk*. Segundo

Coad e Yourdon (1992) a sua principal característica é que o conceito de programar é dividido em duas etapas distintas e independentes, a saber:

- a) Definição das **classes** de objetos do sistema e suas propriedades;
- b) Criação de instâncias das classes definidas, e sua manipulação através da **troca de mensagens** entre os objetos criados.

A maior dificuldade para a compreensão de uma linguagem como o *Smalltalk* reside exatamente na força da cultura tradicional das linguagens algorítmicas, sendo mais assimilável por pessoas que nunca foram expostas às linguagens de programação convencionais.

Ainda segundo Coad e Yourdon (1992) uma linguagem deve possuir quatro elementos de modo a suportar a programação orientada a objetos:

1. **Proteção de dados** - esta característica é importante para assegurar a confiabilidade e capacidade de modificação de sistemas pela redução de interdependências entre os seus componentes. O estado de um módulo de programação deve estar visível somente dentro do escopo do módulo, e os dados só devem ser manipulados por um conjunto localizado de procedimentos. Adicionalmente, uma vez que as variáveis de estado internas a um módulo não sejam acessadas de fora, evita-se efeitos colaterais indesejáveis, desde que a interface entre estes módulos seja projetada com cuidado;
2. **Abstração de dados** - pode ser considerada como o mecanismo que utiliza a proteção de dados. O programador define um tipo abstrato de dados, consistindo de uma representação interna acrescida de um conjunto de procedimentos usados para acessar e manipular dados;
3. **A coesão dinâmica** - é indispensável para utilizar um determinado código para outros tipos de dados. Trata-se do processo em que o chamador recebe o endereço da rotina associando uma mensagem a objetos com o método para executar essa mensagem;
4. **A hereditariedade** - permite que criemos classes e assim podemos criar objetos que são a especialização de outros objetos, uma especialização de uma classe existente é chamada de sub-classe. A classe anterior passa a ser a sua super-classe. Uma sub-classe herda variáveis instanciadas, variáveis de classe e métodos

apropriados a objetos mais especializados. Adicionalmente, uma sub-classe pode ocultar ou fornecer um comportamento adicional aos métodos da super-classe.

Para Coad e Yourdon (1992) enquanto as linguagens procedurais estão voltadas para procedimentos e dados, as linguagens orientadas a objetos estão voltadas para objetos e mensagens, empregam a abordagem de programação centrada em objeto e dado. A proteção e a abstração de dados aumentam a confiabilidade e facilitam a separação de especificações de procedimentos e representação.

A hereditariedade reforça o conceito de fatorização, no qual o código para realizar uma tarefa específica pode ser encontrado em um único lugar, o que facilita sobremaneira a manutenção de software. As bibliotecas de classes pré-definidas beneficiam o desenvolvimento de Software pelo suporte modular ao desenvolvimento de sistemas, resultando em poucas linhas de código e baixa complexidade.

Uma outra linguagem de programação conhecida é a C++ que foi desenvolvida a partir da linguagem procedural 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++.

Um terceiro exemplo de linguagem de programação orientada a objeto é o Object Pascal, que é uma extensão à linguagem procedural 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 Windows que a utiliza como linguagem de programação para o desenvolvimento de aplicações.

Object Pascal suporta completamente os conceitos de Programação orientado a objetos. Permite a definição de objetos, o gerenciamento de objetos por classes e o relacionamento de classes através do mecanismo de herança.

## 6. Escolha da Orientação a Objeto no desenvolvimento de Sistemas

A bibliografia sobre desenvolvimento de sistemas utilizando o paradigma de orientação a objeto começa a ficar cada vez mais rica de casos. Até há alguns anos atrás, a orientação a objeto não passava de uma série de conceitos interessantes mas que careciam de exemplos concretos e que pudessem aprimorar métodos de trabalho.

A transição de uma metodologia convencional para um orientado a objeto deve ser feita de acordo com alguns cuidados e critérios. Segundo Taurion (1997), algumas armadilhas podem comprometer esta transição e são resumidas a seguir:

1. Subestimar o esforço de aprendizado, já que um simples treinamento em uma ferramenta Orientada a Objeto pode não ser suficiente. É importante que se adquiram conceitos sólidos e que se respeite o tempo necessário dentro de uma curva de aprendizado, implicando em investimentos razoáveis;
2. A escolha de ferramentas é importante pois, o *marketing* das mesmas pode levar a enganos. Algumas delas são vendidas como soluções Orientada a Objeto e que permitem implementar soluções corporativas de missão crítica, o que na realidade pode se mostrar injustificado;
3. Também com a escolha infeliz da consultoria pode-se ter graves problemas. Segundo Taurion, “nem sempre uma experiência profunda em C++ ou *Smaltalk* será suficiente para garantir o sucesso de um projeto Orientado a Objeto. Um fator crítico é o gerenciamento do projeto”. Ainda segundo o autor, esta consultoria deve ser escolhida não só pela sua experiência em gerenciamento, mas também através dos sólidos conceitos em Orientação a Objeto, uso de ferramentas e conhecimento do negócio do cliente.

Uma metodologia Orientada a Objeto é tanto mais robusta quanto maior o número de itens dos relacionados a seguir, que sejam adotados. Segundo Pires & Faleiro (1996), são eles:

1. Estar em uso por um período significativo de tempo – cinco anos ou mais – e durante este tempo existiu um processo de melhoria contínua da metodologia em função do uso;
2. Foi utilizada por um número significativo de projetos – em torno de 100 – com objetivos de implementação comercial de aplicações;
3. Um número significativo de referências na literatura técnica, estando documentada em pelo menos um livro disponível amplamente;
4. Suporta mais de uma ferramenta CASE disponível comercialmente;
5. Foi utilizada com sucesso – no sentido amplo – na maioria dos projetos em que foi empregada.

## 7. Comentários e Considerações Finais

O paradigma de orientação a objetos apresenta a característica fundamental de unificar esforços no desenvolvimento de sistemas. Nas seções acima foram abordados os aspectos teóricos da orientação a objetos, com ênfase em metodologias. A orientação a objeto se utilizada em toda a sua potencialidade deverá contribuir na qualidade técnica de projetos e satisfação de clientes e usuários de sistemas. A abordagem do texto procura descrever algumas metodologias sem a preocupação de classificar ou avaliar qual dos métodos é o mais completo, ou se existe uma metodologia melhor que outra, apenas que semelhanças e diferenças devem ser levados em conta quando da adoção de uma delas, para melhor adequação aos trabalhos que se pretende desenvolver.

O que parece ser uma evolução dentre as metodologias apresentadas, é que a UML procura unificar ou pelo menos padronizar os métodos de Booch, Rumbaugh e Jacobson.

Particularmente, recomenda-se ao utilizar os princípios do paradigma da orientação a objetos, a necessidade de desenvolver modelos consistentes de análise e modelagem do negócio como passo prévio e fundamental ao desenvolvimento da programação e definição das bases de dados.

Para se tirar um bom proveito do ambiente de Orientação a Objeto, considera-se alguns aspectos que devem ser observados antes do desenvolvimento completo das aplicações. São eles:

- Definir as metodologias de modelagem, análise e projeto;
- Personalizar as ferramentas de acordo com as metodologias;
- Definir os padrões para a criação de classes de objetos;
- Adaptar os objetos nativos e criar novos objetos;
- Estudar e entender com clareza a estrutura de herança e comportamento das classes de objetos nativos e criados;
- Divulgar os objetos e treinar a equipe de desenvolvimento no seu uso.

A Análise Orientada a Objeto mostra todo o seu potencial caso as fases subseqüentes (modelagem, projeto, programação) também sejam conduzidas

sob o mesmo paradigma. Caso contrário, os mapeamentos e traduções entre as sucessivas fases como, por exemplo, os mapeamentos de modelos de objetos a modelos funcionais ou vice-versa, poderiam transformar o desenvolvimento em algo bastante complexo. Entretanto, para as aplicações mais comerciais onde uma base de dados é indispensável na maioria dos casos, as técnicas de Análise Orientada a Objeto encontram um obstáculo tecnológico: o número de bancos de dados orientados a objeto no mercado ainda é pequeno e com padrões não muito bem definidos, o que significa um mapeamento forçado dos modelos de objetos a modelos relacionais, com a perda da semântica que isto significa, e em particular para a implementação das operações e métodos.

Enquanto a tecnologia de banco de objetos não for adotada em escala, a estratégia para a incorporação da orientação a objeto irá certamente permanecer fundamentada na aplicação da teoria de orientação a objeto durante a fase de modelagem de negócio sucedida pela implementação da base de dados através de bancos de dados relacionais estendidos ou híbridos objeto/relacional. Ficando a cargo dos profissionais de desenvolvimento tirar proveito das similaridades existentes entre os modelos.

Uma técnica ideal de Análise Orientada a Objeto deve:

- Permitir modelagem e especificações precisas e multidimensionais dos sistemas de objetos;
- Organizar os modelos em níveis distintos de abstração e consistentes em todos os aspectos;
- Preferencialmente não forçar o encapsulamento prematuramente;
- Permitir uma transição sem obstáculos às fases de Projeto e Implementação orientados a objetos;
- Facilitar a reutilização dos modelos, minimizando o esforço, permitindo especificar por extensão;
- Permitir validar os modelos através da construção rápida de protótipos;
- Proporcionar critérios para evolução de qualidade dos modelos resultantes de sua aplicação.

A metodologia tradicional de abordagem estruturada de sistemas ainda é bastante utilizada nas instituições. Porém, em reuniões com equipes das áreas de TI do Ministério da Saúde, do Ministério da Educação e consultores de projetos de sistemas, nota-se uma clara tendência de migração para o desenvolvimento orientado a objetos, principalmente com forte treinamento na metodologia UML – *Unified Modelling Language*. No Ministério da Saúde a UML já é uma realidade, tendo sido utilizada no desenvolvimento de um grande sistema interno, implementado e em produção desde 2005. No desenvolvimento de uma nova aplicação, existe uma alta expectativa de reusabilidade de código, implicando em redução de tempo e custo de projeto, sem queda na qualidade dos trabalhos.

A evolução do paradigma de Orientação a Objeto parece se dar de acordo como o esperado. Ainda falta a padronização de alguns conceitos, apesar dos esforços dos diversos grupos organizadores e normativos. A busca de uma maior produtividade em uma área crítica como o desenvolvimento de *software* e a falta de mão de obra qualificada tendem a acelerar a busca de novas ferramentas.

O que precisa ser melhor entendido, é que, se devido a este novo paradigma surgirão novas formas de cooperação, especificamente novas mentalidades e grupos de trabalhos mais eficientes. Este novo modelo requer que se abandone práticas tradicionais, tais como: modelo de desenvolvimento linear (passo a passo), estrutura com funções e responsabilidades fixas e estrutura de poder autocrática. Experimentar novos modelos como o desenvolvimento paralelo, onde múltiplas partes de um mesmo problema são resolvidas simultaneamente, ou de troca de informações no grupo orientadas à ação, devem fazer parte de um estudo mais aprofundado do paradigma da orientação a objetos.

A aceitação de um novo paradigma, principalmente em aprendizado e gerenciamento não significa somente uma rápida troca de geração de código. Deve-se levar em conta os objetivos, políticas, estrutura e *layout* físico. O estudo de casos, tanto bem sucedidos quanto aqueles que resultaram em fracasso, certamente ajudarão a compor o talvez novo cenário cotidiano de desenvolvimento de *software*.

Este novo cenário sugere aos gerentes de projetos ir de uma estrutura com funções e responsabilidades fixas para outra de contribuição, onde se agregam conhecimentos e habilidades. Sugere aos desenvolvedores abandonar os confrontos ficando com a integração, e de troca de apresentações de mão única para formas interativas. Sugere entender realmente o conceito de orientação a objeto, e começar a trabalhar com ele, principalmente para aquelas pessoas que sempre usaram as ferramentas de programação procedural, e aprender a enxergar objetos ao invés de funções e programas.

A Orientação a Objetos necessita mais do que uma nova maneira de pensar. Ela requer uma nova maneira de agir.

## 8. Referências Bibliográficas

BAILIN, S. *An object-oriented requirements specification method*. New York: **Communications of the ACM**, v.32, n.5, p.608-623, may. 1989.

BOOCH, G. *Análisis y diseño orientado a objetos con aplicaciones*. Addison-Wesley. 1996.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML – Guia do usuário*. São Paulo: Editora Campus, São Paulo, 2000.

COAD, P.; YOURDON, E. *Análise baseada em objetos*. Rio de Janeiro: Campus, 1992.

COLEMAN, D. et al. *Desenvolvimento orientado a objetos: o método Fusion* -. Rio de Janeiro, Editora Campus, 1996.

DEMARCO, T. *Análise estruturada e especificação de sistema*. Rio de Janeiro, Editora Campus, 1989.

FICHMAN, R.; KEMERER, C. *Object-oriented and conventional analysis and design methodologies: comparison and critique*. **IEEE Computer**. Los Alamitos, v.25, n.10, p.22-39, Oct. 1992.

GOLENDZINER, L. G. Um modelo de versões para Bancos de dados Orientados a Objetos. Porto Alegre: CPGCC da UFRGS, 1995. Tese de Doutorado.

JACOBSON, I. et al. *Object-oriented software engineering: a use case driven approach*. Reading: Addison-Wesley, 1992.

JACOBSON, I.; GRISS, M.; JONSSON, P. *Software reuse: architecture, process and organization for business success*. Addison-Wesley, New York, 2001. 53 Addison-Wesley, New York, 2001.

JALOTE, P. Functional refinement and nested objects for object-oriented design. **IEEE Transactions on Software Engineering**, New York, v.15, n.3, p.264-270, Mar. 1989.

MARTIN, J.; Odell, J. *Análisis y diseño orientado a objetos*. Prentice Hall. 1994.

MONARCHI, D.; PUHR, G. *A research typology for object-oriented analysis and design*. **Communications of the ACM**. New York, v.35 n.9, p.35-47, Sep. 1992.

OLIVEIRA, J.; CUNHA, C.; MAGALHÃES, G. *Modelo de objetos para construção de interfaces visuais dinâmicas*. In: SIMPOSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE. Recife, 1995.

PIRES, A. Pesquisa DevMag/DRC : O uso da orientação a objetos no Brasil. *Developers' Magazine*, n°12, pp.12, agosto/97.

PIRES, A.; FALEIRO, J. M. Um Comparativo para metodologias de desenvolvimento OO. *Developers' Magazine*, n.4, p.20, dezembro/96.

RUMBAUGH, J. et al. *Modelagem e projetos baseados em objetos*. Editora Campus, Rio de Janeiro, 1994.

SHLAER, S.; MELLOR, S. *Análise de sistemas orientada para objetos*. São Paulo: McGraw-Hill, 1990.

SHLAER, S.; MELLOR, S. *Object life cycles: Modeling the World in states*. Englewood Cliffs: Yourdon Pres, 1992.

TAPSCOTT, D.; CASTON, A. *Mudança de Paradigma*. São Paulo: Makron Books, 1995.

TAURION, C. *Desafio: como migrar para orientação a objetos?*. *Developers' Magazine*, Agosto/97.

TORRES, N. A. *Manual de planejamento de informática empresarial*. São Paulo, Makron Books, 1994.

YOURDON, E. *Object-oriented systems design*. Englewood Cliffs: Prentice-Hall, 1994.