Relatório de Pesquisa CIC/UnB - 05/97

Brasília - DF

Setembro de 1997

Uma avaliação de Métodos Orientados a Objetos e Modelos de Processo

Autoras:

Gilene do Espírito Santo Borges

Prof. Dra. Maria Elenita Menezes Nascimento

Disponível em ftp://ftp.cic.unb.br/pub/publications/report/rr.97-05.ps.Z

Universidade de Brasília.
Instituto Central de Ciências.
Departamento de Ciência da Computação.
Mestrado em Ciência da Computação.
Área: Engenharia de *Software*.
Orientadora: Dra. Maria Elenita Menezes Nascimento.

Uma Avaliação de
Métodos Orientados a Objetos e
Modelos de Processo

Gilene do Espírito Santo Borges.

Setembro - 1997

Mensagem

Nós não gostamos do que não entendemos,
na verdade isto nos assusta.

GASTON.

Agradecimentos

Quero agradecer a **Deus** pela proteção e orientação que me são sempre concedidos.

A minha orientadora, **Dr[a]. Maria Elenita Menezes Nascimento**, pelas críticas, pela orientação e também pelo incentivo que me foram tão preciosos.

Ao colega **M.Sc. Fábio Bianchi Campos** pela grande paciência e inestimável disponibilidade em poder me esclarecer dúvidas a cerca dos aspectos técnicos deste trabalho.

Aos meus pais, **Gil Pereira Borges** e **Leny Cardoso do Espírito Santo Borges**, e irmãos, **Cleide do Espírito Santo Borges** e **Everton do Espírito Santo Borges**, pela ajuda, compreensão e coragem, que mesmo de longe souberam me dar.

A um grande amigo e também namorado, **Alexandre Mesquita Gomes** pelo seu carinho e apoio que me foram tão preciosos, me possibilitando concluir este trabalho com tranqüilidade e perseverança.

Aos meus tios **Manoel Augusto Soares** e **Luceny do Espírito Santo** e primos, que me acolheram em sua casa, me proporcionando uma tranqüilidade inestimável.

Aos colegas de mestrado que, de uma forma ou de outra, me auxiliaram na realização deste trabalho, em especial **M.Sc. Estevam Rafael Hruschka Júnior**, **Renata Peluso de Oliveira** e **Hélio Berchó Pereira**, pela disponibilidade em ajudar-me no entendimento dos aspectos conceituais deste trabalho; no empréstimo de livros e artigos e em como buscar as informações que me eram necessárias.

Aos meus amigos "virtuais": **Carlos Eduardo de Barros Paes** (PUC - São Paulo), **Dr. Guilhermo Bustos Reinoso** (Chile), **Ismar Frango Silveira** (ITA - São Paulo), **Leandro Pompermaier** (UFRGS - Rio Grande do Sul), **M.Sc. Rejane Moreira da Costa** (USP - São Carlos) e **M.Sc. Ricardo Pereira e Silva** (UFRGS - Rio Grande do Sul), que foram muito atenciosos; enviando de longe monografias, referências, artigos e *sites* para que eu pudesse compreender melhor o assunto.

Resumo

Inicialmente, são apresentados os objetivos deste trabalho e a motivação que possibilitou o início e a conclusão deste. São Apresentam-se neste trabalho alguns conceitos na área de engenharia de *software* com o objetivo que o leitor obtenha um esclarecimento destes conceitos básicos, para melhor compreender o que será exposto.

A seguir, os modelos de processos clássicos são descritos, são eles: *waterfall*, prototipação, espiral e transformacional; também suas características, vantagens, desvantagens e aplicabilidade de cada um destes modelos.

Para que o leitor se sinta familiarizado com o paradigma da orientação a objetos, são abordados os principais conceitos desta sub-área. E, posteriormente, os cinco métodos orientados a objetos, escolhidos segundo alguns critérios estabelecidos, são descritos: OMT, OOAD, OOSE, OOA-OOD e FUSION. A descrição dos métodos é um resumo do que o método sugere para ser usado no desenvolvimento de um sistema; as fases a serem seguidas e os modelos e diagramas gerados durante estas fases.

Serão apresentadas também: i) as considerações finais de cada um dos métodos, descrevendo características técnicas e individuais de cada método; ii) uma tabela contendo uma comparação preliminar dos métodos, onde alguns aspectos são identificados em cada um dos métodos; iii) uma tabela nivelando os conceitos utilizados pelos autores dos métodos, pois determinados métodos nomeiam as definições com nomes diferentes dos que geralmente são utilizados; e iv) uma figura ilustrando a cobertura que cada método faz no ciclo de vida do desenvolvimento.

Esta monografia servirá como um *background* para a dissertação de tese, sendo feito uma pesquisa geral na área onde futuramente iremos nos aprofundar.

Sumário

Mensagem ii

Agradecimentos iii

Resumo iv

Lista de Figuras viii

Lista de Tabelas ix

1. Introdução 1

2. Objetivos 2

3. Motivação 3

4. Aspectos Conceituais 4

4.1 Modelos de processo *4*

4.2 Métodos *4*

4.3 Análise de riscos *4*

4.4 Análise de requisitos *5*

4.5 Análise de sistema *5*

4.6 Projeto *5*

4.7 Implementação *5*

4.8 Teste *6*

4.9 Manutenção *6*

4.10 Considerações finais *6*

5. Modelos de Processo 6

5.1 Modelo Waterfall 7

5.2 Modelo Prototipação 9

5.3 Modelo Espiral 13

5.4 Modelo Transformacional 15

5.5 Considerações finais 16

6. Aspectos conceituais referente a orientação a objeto 17

6.1 Objeto 17

6.2 Classe 18

6.3 Atributos e Instâncias 19

6.4 Operações 19

6.5 Encapsulamento 19

6.7 Mensagem 20

6.8 Polimorfismo 20

6.9 Associação e Ligação 20

6.10 Agregação e Herança 21

6.11 Modelo use case 22

6.12 Cenários 23

6.13 Diagrama de eventos 24

6.14 Considerações finais 24

7. Métodos Orientados a Objeto 25

7.1 Escolha dos métodos 26

7.2 Método OOSE 29

7.3 Método OMT 33

7.4 Método OOAD *37*

7.5 Método OOA-OOD *44*

7.6 Método FUSION *48*

7.7 Comparação entre os conceitos *53*

7.8 Comparação entre os métodos *54*

7.9 Cobertura dos métodos *57*

7.10 Considerações finais *58*

8. Conclusões 58

Referências Bibliográficas 60

Lista de Figuras

Figura 1 - Modelo de Processo – *Waterfall* (PRE92) 8
Figura 2 - Modelo de Processo – Prototipação (PRE92) 10
Figura 3 - Modelo de Processo – Espiral (PRE92)...13
Figura 4 - Modelo de Processo – Transformacional (SOM94) 15
Figura 5 - Objetos da classe Polígonos (baseado em MAR94) 18
Figura 6 - Uma classe (BOO91)...18
Figura 7 - Atributos e Instâncias (baseado em SHL90) 19
Figura 8 - O encapsulamento (baseado em MAR94)..20
Figura 9 – Agregação (RUM94) 21
Figura 10 – Herança (MAR94) 22
Figura 11 - Um modelo *use case* (JAC92)...23
Figura 12 - Um cenário para uma chamada telefônica (RUM94).............................24
Figura 13 - Um diagrama de eventos para uma chamada telefônica (RUM94) 25
Figura 14 - Os processos do método *OOSE* (JAC92)..29
Figura 15 - A fase de análise do método *OOSE* (JAC92) 30
Figura 16 - A fase de construção do método *OOSE* (JAC92) 31
Figura 17 - A fase de teste do método *OOSE* (JAC92) 32
Figura 18 - Método OMT (COL96)...34
Figura 19 - Macro processo do método OOAD (BOO94) 38
Figura 20 - Micro processo do método OOAD (BOO94)...41
Figura 21 - O modelo OOA - um modelo multicamada (COA91) 45
Figura 22 - O modelo OOD - um modelo multicamadas e multicomponentes (COA93)..46
Figura 23 - Critérios de adição ao CDP (COA93) 47
Figura 24 - Os componentes do método FUSION (COL96) 49
Figura 25 - As fases do método FUSION..49
Figura 26 - Cobertura dos métodos..57

Lista de Tabelas

Tabela 1 - Maturidade dos métodos básicos orientados a objetos 27

Tabela 2 - Ferramentas CASE que suportam métodos orientados a objetos 28

Tabela 3 - Métodos integradores 28

Tabela 4 - Comparação entre os conceitos utilizados pelos métodos 53

Tabela 5 - Comparação entre os aspectos capturados pelos métodos 55

Tabela 6 - Comparação entre os aspectos capturados pelos métodos (Cont.) 56

1. Introdução

Desde o surgimento da engenharia de *software*, várias maneiras de gerenciar o desenvolvimento de sistemas vem sendo propostas, criando abordagens de desenvolvimento, modelos de processo, métodos, ferramentas automatizadas e gráficas, além de linguagens de programação e métricas, com a finalidade de desenvolver sistemas que garantam qualidade e produtividade [CAM97]. No entanto, os sistemas desenvolvidos continuam excedendo custos e prazos e, muitas vezes, não retratam os requisitos inicialmente definidos pelo usuário [NAS93] e [PRE95].

Uma pesquisa extensiva tem sido feita para tornar o desenvolvimento de sistemas em uma atividade mais produtiva, controlada e efetiva. Muitos métodos, modelos de processo e abordagens surgiram para tentar solucionar os problemas do desenvolvimento, mas cada um analisa o desenvolvimento de uma maneira [NAS90]. Isto explica porquê, por exemplo, um método é tão eficiente no desenvolvimento de um sistema e em outros não.

É difícil acreditar que com tanto ferramental não se consiga desenvolver um sistema com qualidade. Segundo [PRE95] tais problemas ocorrem devido a: (1) o pouco treinamento formal desse novo ferramental pelos gerentes e desenvolvedores e (2) em conseqüência ao primeiro, estes profissionais utilizam este ferramental de maneira incorreta.

Em face aos problemas apresentados, este trabalho visa apresentar um resumo dos principais métodos orientados a objetos e dos modelos de processo clássicos para desenvolvimento de *software*, visando que os gerentes e desenvolvedores de *software* tenham uma melhor compreensão destes aspectos.

O trabalho é dividido em oito sessões, descritas a seguir.

Na seção 2, serão descritos o objetivo geral e os objetivos específicos deste trabalho.

Na seção 3, será descrita qual foi a motivação que fez com que este trabalho fosse realizado.

Na seção 4, são apresentados alguns conceitos de engenharia de *software* para melhor entendimento deste trabalho, como: modelos de processo, métodos, análise de requisitos, análise de sistemas, projeto, codificação, teste e manutenção.

Na seção 5, os modelos de processo clássicos (*waterfall*, prototipação, espiral e transformacional) são apresentados, com sua definição, características, vantagens, desvantagens e aplicabilidade.

Na seção 6, apresentam-se os principais conceitos de orientação a objeto, para uma melhor compreensão dos métodos orientados a objetos que serão apresentados.

Na seção 7, são apresentados os seguintes itens: *i*) a escolha dos métodos; *ii*) os métodos orientados a objetos: OOSE, OMT, OOAD, OOA-OOD e FUSION; sua descrição e considerações finais; *iii*) a apresentação de uma comparação entre os vários conceitos dos métodos; *iv*) uma comparação preliminar entre os métodos descritos; e *v*) a cobertura das fases de desenvolvimento que cada método faz.

Finalmente na seção 8, a conclusão é apresentada onde é feita uma crítica ao trabalho e, logo após as referências bibliográficas, que foram utilizadas para o desenvolvimento deste trabalho.

2. Objetivos

O objetivo geral deste trabalho é apresentar os aspectos gerais dos modelos de processos e dos métodos orientados a objetos.

A partir do objetivo geral de estudar os modelos de processo e os métodos, surgem vários objetivos específicos:

(1) apresentar conceitos na área de engenharia de *software*;

(2) descrever os modelos de processo e suas aplicabilidades;

(3) expor conceitos específicos da orientação a objetos, para uma melhor compreensão já que esta área específica surgiu recentemente[1];

(4) descrever cinco métodos orientados a objetos;

(5) fazer uma comparação preliminar entre os métodos e entre os conceitos utilizados por eles e também, uma explanação sobre a cobertura dos métodos.

Não é objetivo deste trabalho uma descrição de todos os métodos orientados a objetos, uma vez que a descrição de todos eles levaria muito mais tempo do que o disponível para a finalização de uma monografia; e nem tampouco uma comparação ou crítica mais aprofundada dos métodos orientados a objetos, pois este visa somente um levantamento do estado da arte.

Este trabalho servirá como um *background* para a dissertação de tese, que será o próximo trabalho a ser desenvolvido pelas autoras deste. A dissertação terá como objetivo, auxiliar os desenvolvedores de *software* na escolha do melhor método orientado a objeto

[1] [PRE95] classifica o surgimento da tecnologia orientada a objetos na quarta era, a qual iniciou-se na metade da década de 80; sendo assim uma área recente.

(dentre os que são apresentados neste trabalho) para o desenvolvimento de um determinado sistema baseando-se em suas características.

Apresentados os objetivos deste trabalho, segue-se a motivação para a sua realização.

3. Motivação

No início da década de 80 surge uma preocupação com os *softwares* que são desenvolvidos e uma nova compreensão da importância do *software* [PRE95]. Muitos dos problemas relacionados ao desenvolvimento de *software* foram percebidos no final da década de 1960, quando entrou em evidência a expressão "crise do *software*".

Dentre estes problemas podemos citar: (1) baixa produtividade; (2) qualidade abaixo do adequado; (3) prazos e custos estendidos; (4) falta de tempo na coleta dos requisitos do sistema; (5) alto custo de manutenção; e (6) baixa capacidade de previsão; dentre outros. Estes problemas geram insatisfação e falta de confiança por parte dos clientes.

Desde a identificação da crise do *software*, pesquisadores vêm propondo alternativas de solução para tentar minimizar os problemas que afetam o desenvolvimento de *software*. Algumas das alternativas são as seguintes: criação de modelos de processos, métodos, ferramentas automatizadas e gráficas, métricas e linguagens de programação. As alternativas de solução dão uma grande margem para a pesquisa, onde os pesquisadores criam, analisam, comparam, criticam e aplicam para verificar a eficiência.

Foram propostos os métodos estruturados, métodos formais e agora estão sendo propostos os métodos orientados a objetos. Com tantos métodos, surge outro problema: os gerentes de desenvolvimento não sabem como escolher o método mais apropriado para o desenvolvimento de seu sistema, pois à sua frente está um leque de opções; tornando-se necessário o estudo e análise desses novos métodos. Este trabalho visa contribuir com a área de engenharia de *software*, apresentando cinco métodos orientados a objetos bem conceituados (vide tabela 1), bem como considerações finais.

Se torna também necessário o estudo dos modelos de processo clássicos, pois a escolha do método e do modelo de processo necessita ser feita em conjunto, pois seria bastante complicado escolher um modelo de processo que não se adequa a um determinado método. Por exemplo, o uso do modelo de processo prototipação com um método muito complexo não é viável, é necessário o uso de um método mais simples para que a prototipação possa ser utilizada com sucesso.

A seguir serão apresentadas as definições de alguns conceitos mais importantes na área de engenharia de *software* para a compreensão deste trabalho.

4. Aspectos Conceituais

Para que se possa melhor absorver o conteúdo deste trabalho, é necessário a introdução de alguns conceitos, como: modelo de processo, métodos, análise de riscos, análise de requisitos, análise de sistema, projeto, codificação, teste e manutenção.

4.1 Modelos de processo

Sistemas de informação são geralmente desenvolvidos seguindo uma seqüência de estágios, tais como: especificação de requisitos, projeto, implementações e assim por diante. O modelo de processo é visto como uma maneira alternativa de executar essa seqüência de estágios [NAS90].

Os modelos de processo determinam os detalhes dos passos a serem seguidos para se desenvolver um *software*. Eles descrevem o processo e são freqüentemente descartados ou incluídos somente em documentos externos [PRE95].

4.2 Métodos

Métodos de engenharia de *software* proporcionam os detalhes de "como fazer" para construir o *software*. Os métodos envolvem um amplo conjunto de tarefas que incluem: planejamento e estimativa de projeto, análise de requisitos de *software* e de sistemas, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção. Os métodos da engenharia de *software* muitas vezes introduzem uma notação gráfica ou orientada à linguagem especial e introduzem um conjunto de critérios para a qualidade de *software* [PRE95].

4.3 Análise de riscos

Segundo [SOM92], risco é um conceito difícil de definir precisamente. Uma boa maneira para pensar em um risco é simplesmente algo que não pode dar errado. Por exemplo,

se a intenção é usar uma nova linguagem de programação, um risco é que não exista compiladores apropriados disponíveis. Riscos são uma conseqüência de uma informação inadequada e são resolvidos iniciando algumas ações para descobrir informação a qual reduz incerteza.

A análise de riscos é uma série de passos de administração de riscos que nos possibilita "atacar" o risco: identificação, avaliação, disposição por ordem de prioridade, estratégia de administração e monitoração dos riscos [PRE95].

4.4 Análise de requisitos

A análise de requisitos é a fase onde se deve capturar e explicitar os requisitos do usuário a serem satisfeitos [PER96].

O escopo definido para o *software* proporciona uma direção, mas uma definição detalhada do domínio da informação e da função do *software* é necessária antes que o trabalho se inicie. Tal definição é obtida pela análise de requisitos.

4.5 Análise de sistema

A análise de sistema define o papel de cada elemento num sistema baseado em computador, atribuindo, em última análise, o papel que o *software* desempenhará.

Análise é o estágio do ciclo de desenvolvimento no qual um problema do mundo real é examinado para se conhecer seus requisitos sem que se planeje a implementação [RUM94].

4.6 Projeto

O projeto traduz os requisitos do *software* num conjunto de representações (algumas gráficas, outras tabulares ou baseadas em linguagem) que descrevem a estrutura de dados, a arquitetura, o procedimento algorítmico e as características de interface [PRE95].

4.7 Implementação

As representações do projeto devem ser convertidas numa linguagem artificial (uma linguagem de programação convencional; ex.: C++, Eiffel, Visual Basic, Access), que resulte em instruções que possam ser executadas pelo computador [PRE95].

4.8 Teste

Logo que é implementado numa forma executável por máquina, o *software* deve ser testado para que se possa descobrir defeitos de função, lógica e implementação [PRE95].

Jacobson [JAC92], sugere que o teste seja dividido em: *i*) teste dos módulos separadamente; *ii*) teste de integração destes módulos; e *iii*) teste do sistema como um todo.

4.9 Manutenção

Na fase de manutenção, concentram-se as mudanças que estão associadas à correção de erros, adaptações exigidas à medida que o ambiente do *software* evolui e ampliações produzidas por exigências variáveis do cliente [PRE95].

Rumbaugh [RUM94] acrescenta dizendo que esta fase será a de menor importância no desenvolvimento de *software*, pois o esforço deve ser na análise e projeto.

4.10 Considerações finais

Os últimos sete conceitos (análise de requisitos, análise de risco, análise, projeto, codificação, teste e manutenção) são fases do ciclo de desenvolvimento, que deveriam ser cobertas por todos os métodos para que estes fossem completos; porém veremos no decorrer do trabalho, que poucos métodos cobrem mais fases do que as essencialmente necessárias[2]: análise, projeto e implementação.

Agora que já temos os conceitos da área de engenharia de *software* estabelecidos, serão apresentados os modelos de processo.

5. Modelos de Processo

Os modelos de processo estão preocupados em estabelecer a ordem de estágios envolvidos no desenvolvimento e evolução do *software* e em definir os critérios de transição de uma fase para outra. Tais modelos são importantes, porque eles sugerem a ordem na qual os projetos devem executar suas fases maiores [NAS92].

O provérbio na parede da sala do meu professor na minha escola "por que nunca existe tempo suficiente para fazer as coisas certas da primeira vez, mas sempre existe bastante tempo para fazê-lo novamente?" ...

Jessica Keyes

Nos anos 60, o profissional liberal na arte de desenvolver sistemas começa a reconhecer que existia (ou deveria existir) um método para se fazer tal tarefa. Não está certo se o conceito apareceu espontaneamente ou cresceu gradualmente (a literatura está cheia de opiniões de todo jeito), mas em diferentes áreas, aproximadamente ao mesmo tempo, indivíduos, pesquisadores, empresas e agências governamentais começaram a pensar sobre desenvolver sistema de uma maneira mais organizada [KEY93].

Surgiram, então, modelos de processo preocupados em apresentar etapas pelas quais o desenvolvimento de *software* deveria se enquadrar, devido as limitações destes, foram sendo criados, posteriormente, outros modelos, os quais eram a combinação ou extensão dos anteriores.

Com esta visão foram definidos vários modelos de processo; aqui serão apresentados os quatro modelos mais reconhecidos da área, são eles: *waterfall*, prototipação, espiral e transformacional, com o intuito de esclarecer profissionais a organizarem o desenvolvimento de *software*.

Veremos também nesta seção, definições, características, vantagens, desvantagens e aplicabilidades dos modelos de processo citados anteriormente.

5.1 Modelo *Waterfall*

5.1.1 Definição do modelo waterfall

O modelo de processo *waterfall* foi desenvolvido por Royce [ROY70] para superar os problemas de desenvolver grandes *softwares*. Neste modelo, o *software* é desenvolvido em sucessivos estágios, os quais são ilustrados na figura 1. *Loops* de realimentação entre as etapas são recomendados como um meio de verificar a corretude e integração entre as fases [NAS90].

---

[2] Essencialmente necessárias, quer dizer, o que geralmente é utilizada durante o desenvolvimento de um sistema.

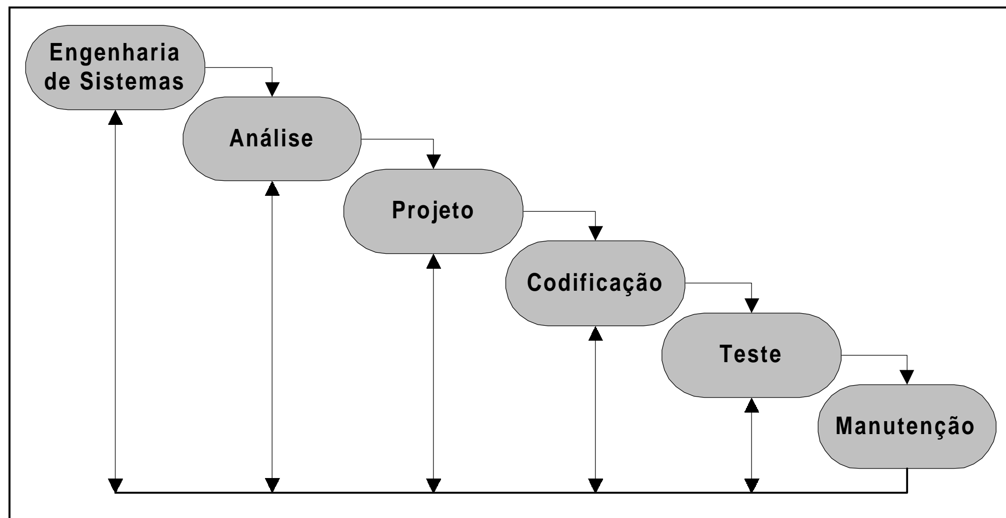

Figura 1 - Modelo de Processo – *Waterfall* (PRE92)

O modelo *waterfall*, ou modelo cascata, é o paradigma do ciclo de vida clássico. Este modelo requer uma abordagem sistemática[3] e seqüencial ao desenvolvimento do *software*, que se inicia no nível do sistema e avança ao longo da análise, projeto, codificação, teste e manutenção [PRE92].

5.1.2 Características do modelo waterfall

O modelo *waterfall* tem um lugar definido e importante na engenharia de *software*. Ele produz um padrão no qual os métodos para análise, projeto, codificação, testes e manutenção podem ser colocados. Além disso, as etapas deste modelo são muito semelhantes às etapas genéricas dos outros modelos de processo [PRE92].

Este modelo se caracteriza pelo fato de que a etapa seguinte só se inicia, quando a etapa anterior tiver sido concluída. Por isto, é um modelo de processo procedimental.

5.1.3 Vantagens do modelo waterfall

[NAS90] apresenta algumas vantagens em utilizar o modelo *waterfall* para o desenvolvimento: (1) as etapas sucessivas usadas no modelo *waterfall* ajudam a eliminar muitas das dificuldades originalmente encontradas nos projetos de *software*, tais como manutenção cara e programas pobremente estruturados; e (2) o modelo também fornece um ambiente mais gerenciável para o desenvolvimento de *software* distinguindo definições de

[3] O mesmo que ordenada, metódica [FER77].

requisitos de projeto e implementação, permitindo assim, uma maneira mais eficiente de planejar e controlar o desenvolvimento do *software*.

5.1.4 Desvantagens do modelo waterfall

[SOM92] e [PRE92] concordam que o modelo *waterfall* é o mais antigo e largamente usado, mas tem sido bastante criticado devido aos problemas que, às vezes, surgem quando ele é utilizado, que são: (1) requisitos do sistema devem ser bem definidos, nem sempre isso é possível no começo do desenvolvimento; (2) não existe interação entre desenvolvedor e cliente, então, uma versão do programa só será entregue bem mais tarde e encontrar um erro pode ser desastroso; e (3) [PRE92] acrescenta que os projetos reais raramente seguem o fluxo seqüencial que o modelo propõe, sempre há a inclusão de novos requisitos trazendo problemas.

5.1.5 Aplicabilidade do modelo waterfall

Aqui será apresentado onde o modelo *waterfall* é aplicado com êxito e onde não se deve empregá-lo.

O modelo *waterfall* pode ser o modelo de desenvolvimento mais apropriado:

i) onde a organização necessita desenvolver um grande sistema para ser mais estruturado e gerenciável [KAN95];

ii) onde existe o desenvolvimento de sistemas como compiladores e sistemas operacionais [NAS90].

[NAS90] acrescenta que o uso do modelo seria inadequado, quando não é possível construir uma completa descrição dos requisitos e descrições intermediárias detalhadas de cada fase.

5.2 Modelo Prototipação

5.2.1 Definição do modelo prototipação

A prototipação é uma técnica para ajudar a estabelecer e validar os requisitos do sistema [SOM92] e [PRE95].

A prototipação é um processo que capacita o desenvolvedor a criar um modelo do *software* que será implementado. O modelo pode assumir uma das três formas: (1) um protótipo baseado em computador que retrata a interação homem-máquina de uma forma que capacita o usuário a entender quanta interação ocorrerá; (2) um protótipo que implementa algum subconjunto da função exigida do *software* desejado; ou (3) um programa existente que executa parte ou toda a função desejada, mas que tem outras características que serão melhoradas em um novo esforço de desenvolvimento [PRE92].

A seqüência de eventos para o modelo de processo prototipação é ilustrado na figura 2.

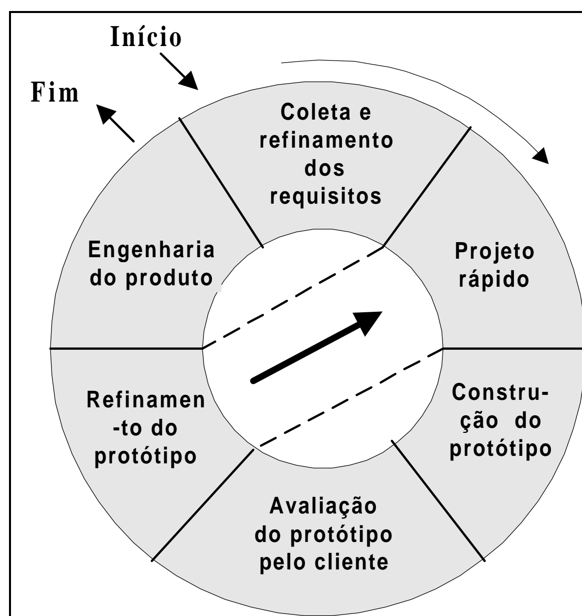

Figura 2 - Modelo de Processo – Prototipação (PRE92)

5.2.2 Características do modelo prototipação

A primeira fase do desenvolvimento utilizando prototipação envolve desenvolver um programa para o usuário experimentar. O objetivo do desenvolvimento é estabelecer os requisitos do sistema. Este é seguido pela reimplementação do *software* para produzir um sistema com qualidade [SOM92].

Evidências mostram que desenvolver um produto através da prototipação é substancialmente mais rápido e mais barato do que os métodos tradicionais. [HUM89] aponta algumas considerações no uso de prototipação:

i) se somente se pretende que seja um experimento rápido, os objetivos do protótipo devem ser claramente estabelecidos antes de começar a construí-lo;

ii) definir o processo de prototipação. Prototipação não significa desenvolver o código da maneira que ele surge na cabeça. Os métodos usados devem ser apropriados para os objetivos específicos do protótipo;

iii) os resultados da prototipação devem ser documentados e analisados antes da decisão ser feita em como proceder;

iv) completar o protótipo inicial e analisar os resultados antes de começar outra iteração do protótipo. As iterações não planejadas podem ser caras;

v) não existe uma maneira certa de prototipar. O resultado pode ser jogado fora, usado como uma base para melhoramento, ou reempacotado como produto. Tudo depende dos objetivos originais, o processo usado e a qualidade desejada do resultado.

Booch aponta em [BOO96], protótipos devem ser criados exclusivamente por razões nobres, das quais existem poucas:

i) obter um entendimento mais profundo sobre os requisitos de um sistema;

ii) determinar se certa funcionalidade, performance, custo, confiança ou adaptabilidade podem ser realizadas;

iii) avaliar o risco associado com certo projeto;

iv) servir a equipe de desenvolvimento como veículo para comunicar um projeto ao cliente;

v) servir como ferramenta de aprendizagem para a equipe de desenvolvimento.

5.2.3 Vantagens da prototipação

As vantagens de utilizar a prototipação, segundo [BOE84] [NAS90], são: (1) produtos com melhores interfaces homem-máquina; (2) sempre tem algo que funciona; (3) reduz o efeito do prazo de entrega no final do projeto; e (4) [HUM89] acrescenta que o desenvolvimento é mais rápido e mais barato do que os métodos tradicionais.

Acredita-se que o surgimento destas vantagens se derivam do fato de que desde o começo do desenvolvimento do sistema, usando-se o modelo de processo prototipação, existe uma interação forte entre o cliente e o desenvolvedor.

5.2.4 Desvantagens do modelo prototipação

Apesar de ser um modelo que apresenta várias vantagens, também possui suas desvantagens e dois autores apresentam diferentes visões sobre estas desvantagens.

Segundo [BOE84] em [NAS90], as desvantagens de utilizar a prototipação são: (1) proporcionalmente menos esforço em planejamento e projeto e mais em teste e concerto; (2) mais dificuldade em integração devido a falha de especificação de interface; e (3) projeto menos coerente[4].

Segundo [PRE92], existem duas desvantagens em se utilizar a prototipação, são elas: (1) quando se consegue realmente definir os requisitos do sistema com a ajuda do usuário, um protótipo foi criado e se parecerá, ao modo de ver do usuário, com o produto final, então o usuário não deseja que este protótipo seja descartado para a criação de um novo produto que levará tempo e dinheiro, mesmo que este seja mais eficiente, e o maior problema é que na maioria das vezes a equipe de desenvolvimento cede; (2) acontece quando a equipe de desenvolvimento adota opções como um sistema operacional ou linguagem de programação impróprios, pois estão a disposição, ou algoritmos ineficientes para demonstrar capacidade e se esquecem que estas opções foram adotadas para a rápida prototipação e as englobam ao sistema.

5.2.5 Aplicabilidade do modelo prototipação

Foram encontradas diversas situações onde o modelo prototipação seria empregado com sucesso, são elas:

i) quando o usuário definiu um conjunto de objetivos gerais para o *software*, mas não identificou os requisitos em detalhes [PRE92];

ii) quando o desenvolvedor pode não ter certeza da eficiência de um algoritmo, da adaptabilidade de um sistema operacional [PRE92];

iii) em situações onde o projeto do produto é complexo e não estruturado [NAS90];

iv) quando os riscos da interface do usuário são dominantes [SOM92], isto significa que a interface é muito importante no sistema, então com o uso da prototipação pode ser verificado se a interface está sendo modelada de maneira correta.

---

[4] Acredita-se que é devido a falta de especificação bem definida, o projeto nem sempre se encontra coerente com os requisitos do usuário.

5.3 Modelo Espiral

5.3.1 Definição do modelo espiral

O modelo espiral para a engenharia de *software* foi desenvolvido para abranger as melhores características tanto do ciclo de vida clássico como da prototipação, acrescentando, ao mesmo tempo, um novo elemento - a análise de riscos - que falta aos anteriores. O modelo, representado pela figura 3, define quatro importantes atividades representadas pelos quatro quadrantes da figura: (1) planejamento; (2) análise de riscos; (3) engenharia; e (4) avaliação feita pelo cliente [PRE92].

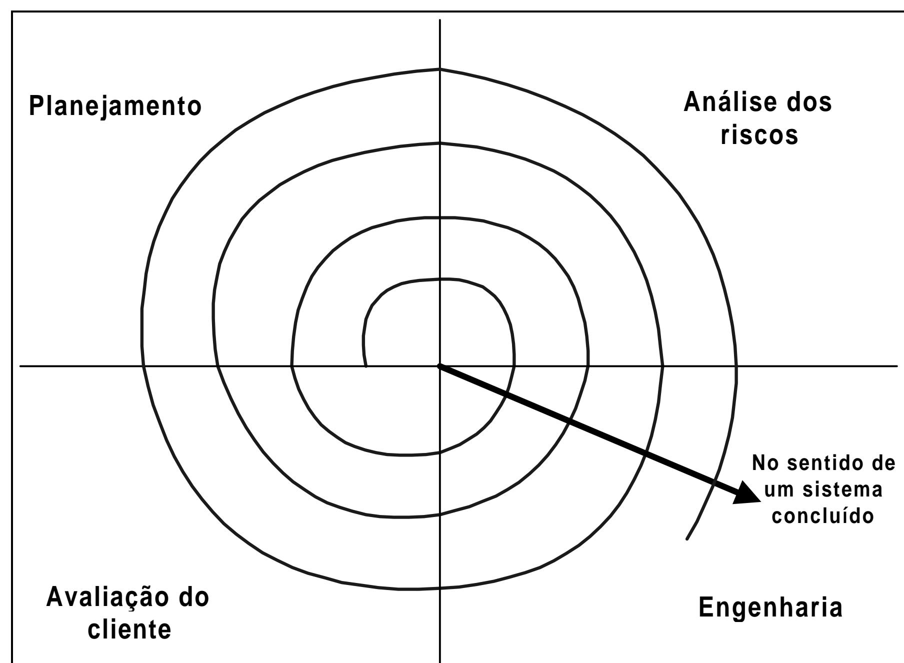

Figura 3 - Modelo de Processo – Espiral (PRE92)

Segundo [JUL88] em [NAS90], o modelo espiral tenta prever e evitar riscos ao invés de só certificar se o produto produzido pela fase de desenvolvimento corrente concorda com as especificações emitidas pela fase anterior.

5.3.2 Características do modelo espiral

Segundo [GIL88] em [PRE92], o modelo espiral usa uma abordagem "evolucionária" à engenharia de *software*, capacitando o desenvolvedor e o cliente a entender e reagir aos riscos em cada etapa evolutiva. O modelo espiral usa a prototipação como um mecanismo de

redução dos riscos, mas, o que é mais importante, possibilita que o desenvolvedor aplique a abordagem de prototipação em qualquer etapa da evolução do produto. Ele mantém a abordagem de passos sistemáticos sugerida pelo ciclo de vida clássico, mas incorpora-a numa estrutura iterativa que reflete mais realisticamente o mundo real.

Segundo [SOM92], sua característica chave é uma avaliação do gerenciamento dos itens de riscos em etapas regulares no projeto e a iniciação de ações para agir contra estes riscos. Antes de cada ciclo, uma análise de riscos é iniciada e no final de cada ciclo, um procedimento de revisão avalia se passa para o próximo ciclo do espiral.

5.3.3 Vantagens do modelo espiral

A grande vantagem do modelo espiral, é que neste modelo existe um processo contínuo de reavaliação, possibilitando cliente e desenvolvedor a: (1) identificarem os riscos em cada fase do ciclo; (2) tomarem decisões para reagirem contra os riscos; e (3) decidirem se continuam ou não o desenvolvimento.

5.3.4 Desvantagens do modelo espiral

As principais desvantagens em se utilizar o modelo espiral são: (1) pode ser difícil convencer grandes clientes de que a abordagem evolutiva é controlável; (2) ela exige considerável experiência na avaliação dos riscos e fia-se nessa experiência para o sucesso; e (3) se um grande risco não for descoberto, indubitavelmente ocorrerão problemas.

5.3.5 Aplicabilidade do modelo espiral

O modelo espiral possui algumas aplicabilidades, que são apresentadas a seguir, segundo [NAS90]:

i) aplicável em desenvolvimento de sistemas para reduzir incerteza;

ii) aplicável em sistemas muito grandes, porque o modelo fornece um mecanismo mais completo de assegurar os requisitos;

iii) aplicável em sistemas altamente distribuídos[5], devido a característica do modelo de análise de riscos.

5.4 Modelo Transformacional

5.4.1 Definição do modelo transformacional

Segundo [NAS90] e [SOM92], a abordagem transformacional, apresentada na figura 4, envolve desenvolver uma especificação formal do sistema e transformar esta especificação em um programa. Entretanto, o progresso entre estes dois pontos é feito aplicando uma série de transformações (T1, T2, T3 e T4), que produzirão sucessivas versões até que o programa final seja desenvolvido. Idealmente, todas as regras de transformação[6] (R1, R2 e R3) preservam corretude, tal que o produto final satisfaça a especificação original.

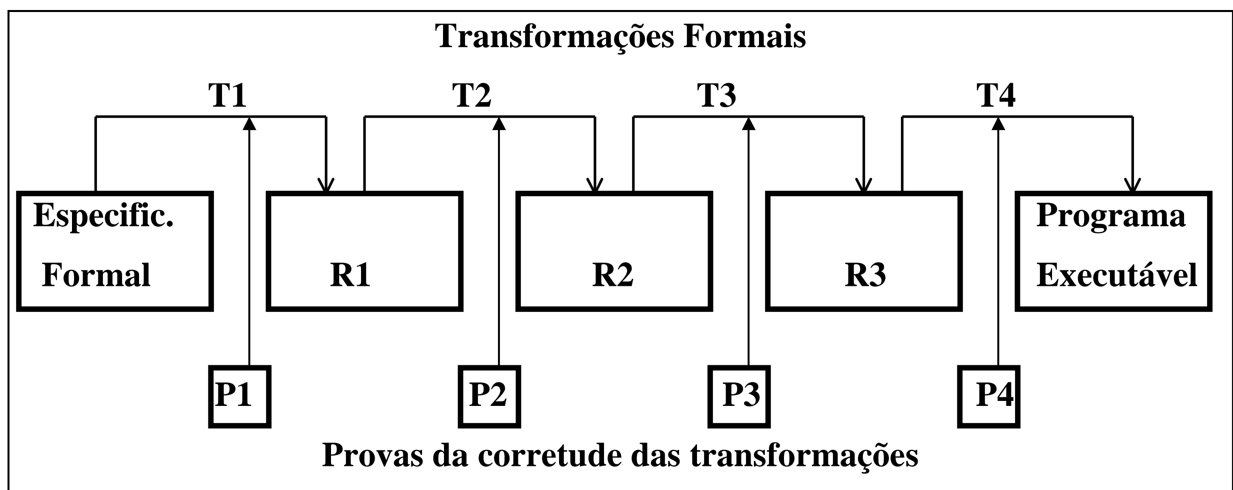

Figura 4 - Modelo de Processo – Transformacional (SOM94)

5.4.2 Vantagens do modelo transformacional

O modelo transformacional evita a dificuldade de ter que modificar o código o qual tem se tornado pobremente estruturado através das repetidas otimizações, pois as modificações e ajustamentos são feitos nas especificações [NAS90].

---

[5] Sistemas distribuídos é o processamento no qual parte ou todo o processamento, armazenamento, funções de controle e funções de entrada e saída estão situados em diferentes locais e conectados entre si através de recursos de transmissão [CAM93].

[6] Transformação pode envolver especialização ou generalização da representação corrente, escolhendo uma estrutura de dados, mudando um controle de estrutura para melhorar a eficiência ou outras alterações [NAS90].

[NAS90] apresenta as vantagens da aplicação deste modelo de processo:

i) automação da aplicação de transformações, as quais reduzem o trabalho intensivo do desenvolvimento de *software*;

ii) segurança em aplicar transformações que podem ser expressas formalmente para assegurar que elas preservam corretude e são livres dos efeitos colaterais;

iii) a fase de teste do produto final pode ser substancialmente reduzida ou virtualmente eliminada em muitos casos, o qual pode ser parcialmente substituído pela verificação das especificações formais [AGR86].

Segundo [SOM92], quando se pretende provar que o programa está de acordo com suas especificações, o uso do modelo transformacional é melhor, pois a distância entre cada transformação é menor do que a distância entre a especificação e o programa. Provas de programas são muito grandes e impraticáveis para sistemas de grande escala, mas a abordagem transformacional cria uma seqüência de passos menores para ser mais eficiente.

5.4.3 Desvantagens do modelo transformacional

Segundo [SOM92], escolher qual transformação aplicar é uma tarefa prática (exige conhecimento) e provar a correspondência das transformações é difícil.

5.4.4 Aplicabilidade do modelo transformacional

Segundo [SOM92], o modelo transformacional é usado quando os riscos de segurança são a principal consideração no processo de desenvolvimento.

O modelo transformacional é uma abordagem muito poderosa, mas pode somente ser aplicada a áreas muito específicas da ciência da computação. Este modelo de processo apresenta dificuldades quando aplicada a sistemas instáveis e pobremente estruturados e orientados a requisitos difíceis de serem expressos formalmente [NAS90].

5.5 Considerações finais

A partir do que foi exposto nesta seção sobre os modelos de processo, pode-se notar que cada um dos modelos apresentados abordam diferentes características do *software* a ser desenvolvido. Por exemplo: o *waterfall* é usado para sistemas onde os requisitos são bem definidos, sendo um modelo bastante metódico; a prototipação é utilizada para sistemas onde

há a necessidade de definir os requisitos e de um desenvolvimento mais rápido e barato; o modelo espiral é utilizado quando é necessário a análise de riscos para decidir se deve continuar o desenvolvimento ou não; e o modelo transformacional aplicado a sistemas bem estruturados com requisitos expressos formalmente.

Como cada um desses modelos abordam diferentes características do *software*, o estudo e compreensão desses modelos são muito importantes, pois sua escolha é vital para o sucesso ou fracasso do desenvolvimento.

Com os modelos de processo já apresentados, serão apresentados agora os conceitos de orientação a objeto com a finalidade que o leitor/pesquisador possa compreender melhor os métodos orientados a objetos que serão descritos logo após os conceitos.

6. Aspectos conceituais referente a orientação a objeto

Para melhor compreensão dos métodos orientados a objetos que serão apresentados a seguir, são apresentados aqui alguns conceitos relacionados a orientação a objeto, como: objeto, classe, atributos e instâncias, operações, encapsulamento, mensagem, polimorfismo, associação e ligação, agregação e herança; e outros relacionados aos métodos que serão apresentados: atores, *use cases*, modelo *use case*, cenários e diagrama de eventos.

6.1 Objeto

Um objeto representa um ‘elemento’ que pode ser identificado de maneira única. Em nível apropriado de abstração[7], praticamente tudo pode ser considerado como objeto. Assim, elementos específicos como pessoas, organizações, máquinas ou eventos podem ser considerados como objetos [COL96].

A figura 5, ilustra a definição de objetos, que segundo [SHL90], é a abstração de um conjunto de coisas semelhantes.

Um objeto tem estado, comportamento e identidade; a estrutura e comportamento de objetos similares são definidos em suas classes comuns [BOO91].

[7] Abstração é o exame seletivo de determinados aspectos de um problema. O objetivo da abstração é isolar os aspectos que sejam importantes para algum propósito e suprimir os que não o forem. A abstração deve sempre visar a um propósito, porque este determina o que é e o que não é importante.

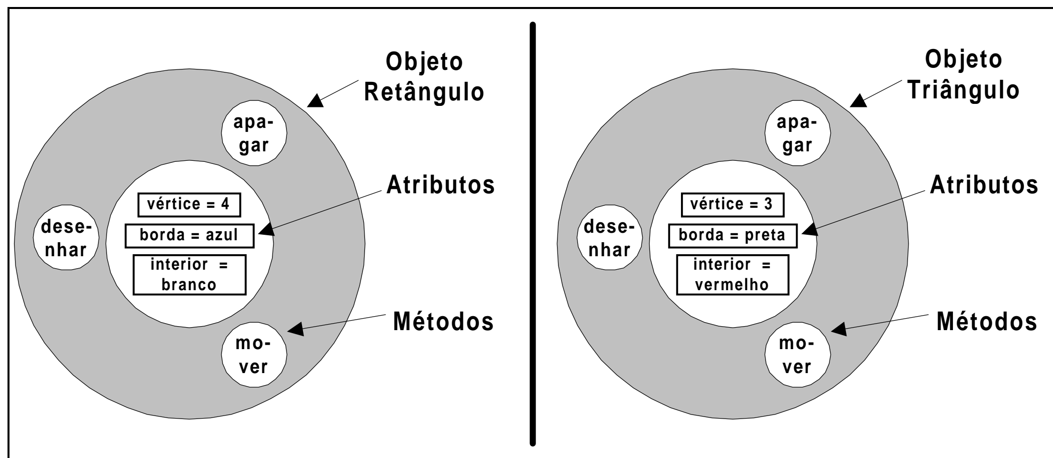

Figura 5 - Objetos da classe Polígonos (baseado em MAR94)

6.2 Classe

Segundo [BOO91], uma classe é um conjunto de objetos que compartilham uma estrutura comum (atributos) e um procedimento comum (operações); ilustrado na figura 6.

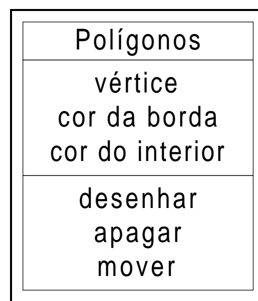

Figura 6 - Uma classe (BOO91)

Uma classe abstrata é uma classe que não possui instâncias diretas mas cujas descendentes, sim. Uma classe concreta é uma classe instanciável; isto é, pode ter instâncias diretas. Uma classe direta pode ter subclasses abstratas (mas estas, por sua vez, devem possuir descendentes concretos) [RUM94].

As classes são organizadas em níveis hierárquicos compartilhando estruturas e comportamentos comuns e são associadas a outras classes. As classes definem os valores de atributos relativos a cada instância de objetos e as operações que cada objeto executa ou a que se submete [RUM94].

Assim, cada objeto é dito ser uma instância de sua classe.

6.3 Atributos e Instâncias

Os atributos são os dados escondidos pelo objeto em uma classe. Cada atributo tem um valor para cada instância do objeto. Este valor deve ser um valor de dado puro, não um objeto [BAK96].

A figura 7, ilustra os conceitos já mencionados de atributo e instância.

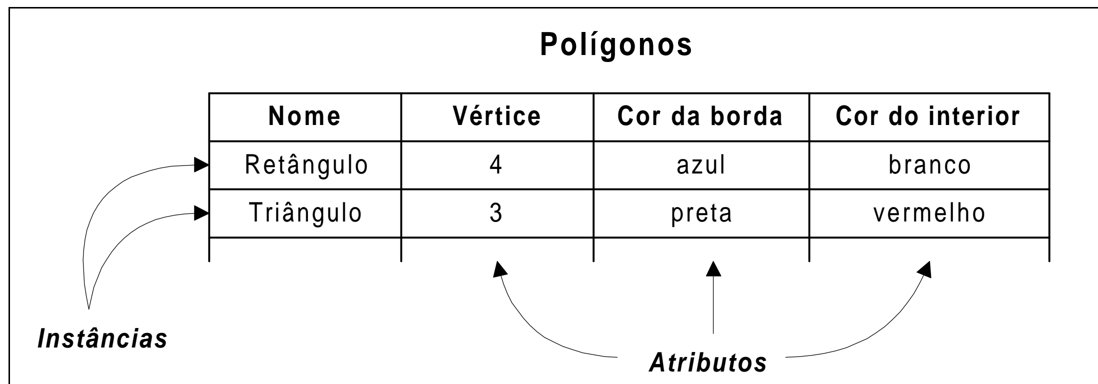

Figura 7 - Atributos e Instâncias (baseado em SHL90)

6.4 Operações

Segundo [BAK96], operações ou métodos são as funções ou transformações que podem ser aplicados em ou por um objeto numa classe. Todos os objetos numa classe compartilham as mesmas operações. Cada operação tem um objeto alvo como um argumento implícito. Uma operação pode ter argumentos em adição ao seu objeto alvo, os quais parametrizam a operação.

A operação é parte da classe e não parte do objeto. A operação pode não ser parte da classe; mas ser uma herança de uma superclasse na hierarquia de classes [MAR94].

6.5 Encapsulamento

O encapsulamento (ilustrado na figura 8), é justamente o empacotamento dos atributos e das operações numa mesma classe. Isto protege os dados contra corrupção, pois somente as operações da classe poderão alterar as estruturas de dados desta classe em questão.

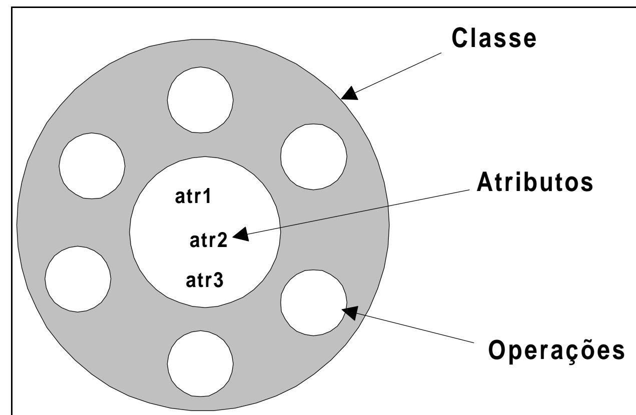

Figura 8 - O encapsulamento (baseado em MAR94)

6.7 Mensagem

Os objetos se comunicam entre si através da mensagem. A mensagem especifica que uma determinada operação de um objeto necessita utilizar uma ou mais operações de outro objeto. Podem ser passados objetos como parâmetro, e, opcionalmente, algum resultado ou valor pode ser retornado. As mensagens especificam que as operações devem ser executados, não como estas operações devem ser executados [MON94].

6.8 Polimorfismo

Segundo [MON94], polimorfismo é a capacidade de um objeto enviar uma mensagem genérica para muitos outros objetos. Cada objeto alvo implementa a operação, que atende a mensagem genérica, de diferentes maneiras.

Método polimórfico[8] significa que a mesma operação toma diferentes formas em diferentes classes.

6.9 Associação e Ligação

A associação é o relacionamento entre classes e a ligação é o relacionamento entre objetos. Assim, uma ligação é uma instância de uma associação [RUM94]. As associações são bidirecionais e podem ser binárias, ternárias ou de ordem mais elevada; na prática, geralmente são binárias.

---

[8] Por exemplo: a classe Arquivo pode ter a operação imprimir para diferentes tipos de arquivos (texto, gráfico, arquivos ASCII). Todos estes métodos executam logicamente a mesma tarefa – imprimir arquivo; assim podemos referir-nos a eles pela operação genérica imprimir.

6.10 Agregação e Herança

Agregação e herança são tipos de associação, onde a herança é também chamada por outros autores como generalização e especialização[9].

A agregação é o relacionamento “parte-todo” ou “uma-parte-de” no qual os objetos que representam as partes, são associados a um objeto que representa o todo. Ou seja, as partes formam o todo. A figura 9 ilustra o conceito de agregação.

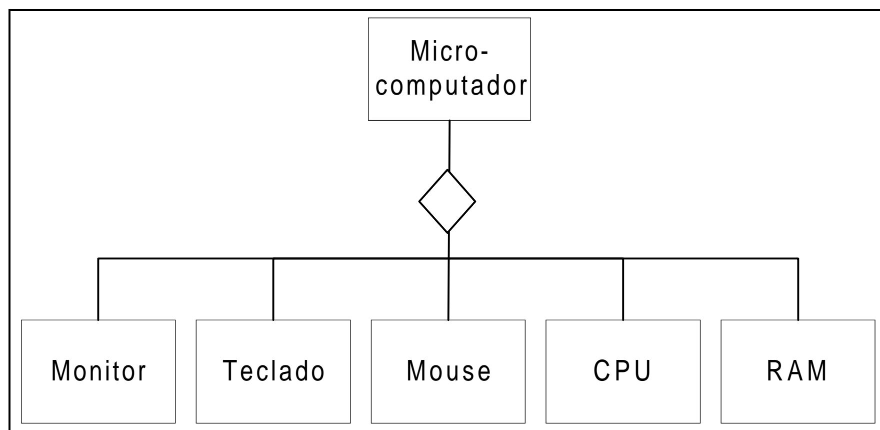

Figura 9 – Agregação (RUM94)

Segundo [MAR94], a herança de classe torna possível que uma classe compartilhe os atributos e as operações de outra classe (superclasse). Mas essa classe que herda, também tem suas operações e, às vezes, atributos próprios.

Na herança simples, uma classe pode herdar os atributos e as operações de uma superclasse; e na herança múltipla, uma classe pode herdar os atributos e as operações de mais de uma superclasse. As três definições são ilustradas na figura 10.

Segundo [RUM94] a vantagem da herança múltipla é maior capacidade de especificação de classes e a maior oportunidade de reutilização; e a desvantagem é a perda da simplicidade conceitual e de implementação.

[9] Utiliza-se generalização para nos referirmos ao relacionamento entre classes, enquanto herança refere-se ao mecanismo de compartilhamento de atributos e métodos utilizando o relacionamento de generalização [RUM94].

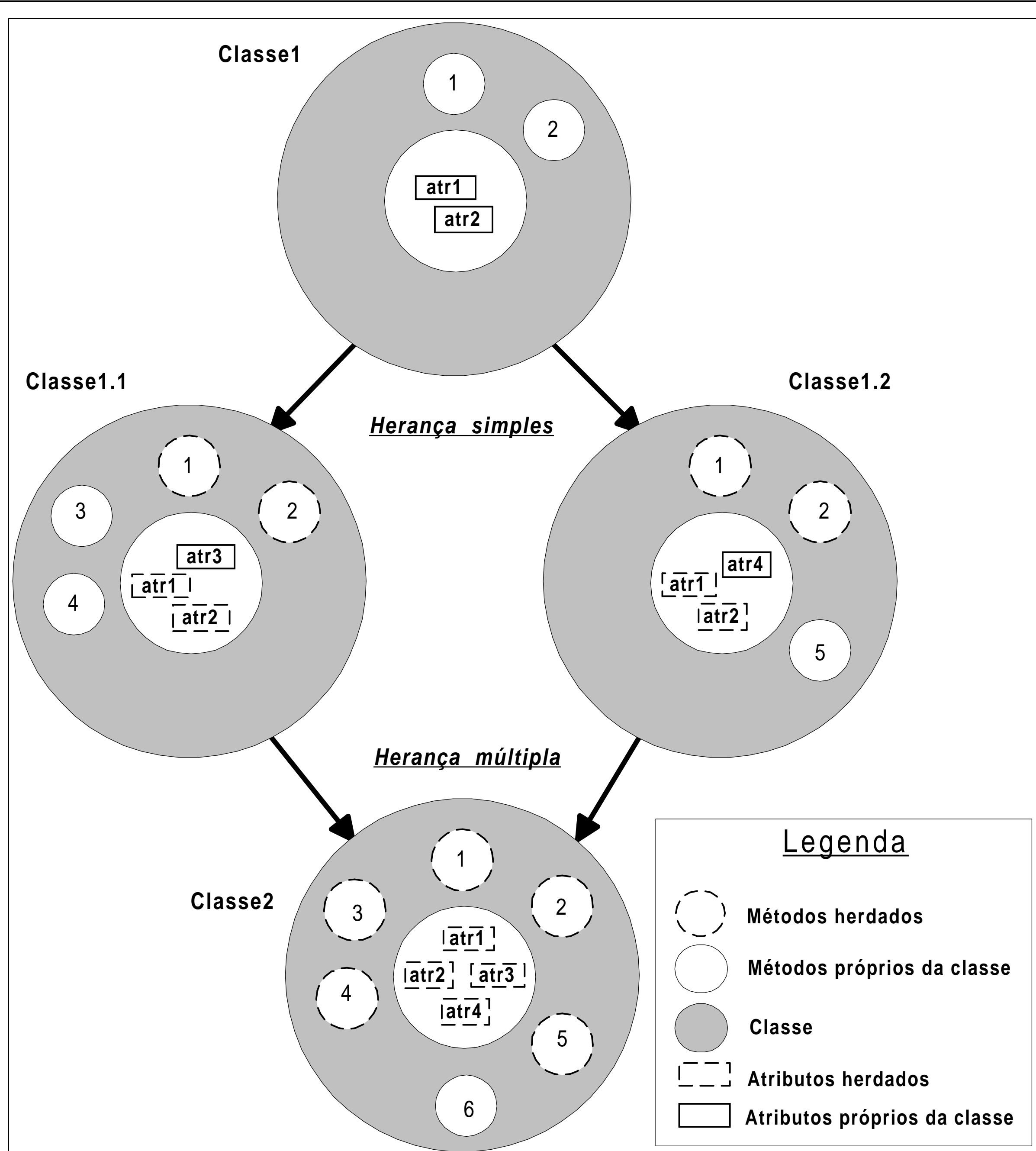

Figura 10 – Herança (MAR94)

6.11 Modelo *use case*

Este conceito, por ser utilizado em mais de um método orientado a objeto (OOSE e OOAD), que serão apresentados na próxima seção, merece ser descrito aqui.

O modelo *use case* compõe-se de atores e *use cases*. Estes conceitos são simplesmente uma ajuda para definir o que existe do lado de fora do sistema (atores) e que deve ser executado pelo sistema (*use case*) [JAC92].

Atores modelam algo que necessita trocar informação com o sistema. Atores podem modelar usuários humanos, mas eles podem também modelar outros sistemas que comunicam com o que está sendo modelado.

Cada *use case* constitui um percurso completo de eventos iniciados por um ator e ele especifica a interação que acontece entre um ator e o sistema. O *use case* é bastante parecido com o diagrama de eventos de Rumbaugh, ilustrado na figura 13.

Os conceitos de atores e *use case*, os quais compõem o modelo *use case*, estão ilustrados na figura 11.

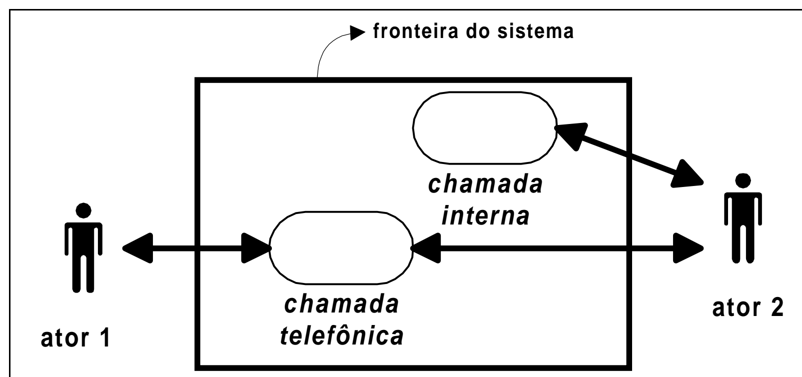

Figura 11 - Um modelo *use case* (JAC92)

6.12 Cenários

O cenário é um conceito também utilizado por mais de um método merecendo ser descrito.

Segundo [RUM94], cenário é uma seqüência de eventos que ocorrem durante uma determinada execução de um sistema. A abrangência de um cenário pode variar; ele pode incluir todos os eventos do sistema, ou somente aqueles que influenciam ou que são gerados por certos objetos do sistema. Um exemplo de um cenário para uma ligação telefônica, é ilustrada na figura 12.

```
chamador levanta receptor
sinal de discar começa
chamador disca dígito (5)
sinal de discar pára
chamador disca dígito (5)
chamador disca dígito (5)
chamador disca dígito (1)
chamador disca dígito (2)
chamador disca dígito (3)
chamador disca dígito (4)
telefone chamado começa a tocar
ouve-se o tilintar do telefone chamando
pessoa chamada atende
telefone chamado pára de tocar
som de chamada desaparece do telefone chamador
telefones são interligados
pessoa chamada desliga
telefones são desligados
chamador desliga
```

Figura 12 - Um cenário para uma chamada telefônica (RUM94)

6.13 Diagrama de eventos

Após a escrita do cenário é necessário identificar os objetos remetente e recebedor de cada evento. A seqüência de eventos e os objetos que permutam eventos podem ser mostrados em um cenário ampliado denominado diagrama de eventos, o qual é ilustrado na figura 13.

6.14 Considerações finais

A partir da apresentação dos conceitos de orientação a objeto, pode-se notar que as idéias são bem diferentes daquelas introduzidas pelos métodos estruturados. O paradigma de orientação a objetos é uma evolução do paradigma estruturado e vem com o intuito de melhorar o desenvolvimento de *software* abordando-o de outra maneira, visando principalmente o reuso com o uso de objetos, herança e encapsulamento.

Vejamos agora os cinco métodos orientados a objeto (OMT, OOAD, OOSE, OOA-OOD e FUSION), comparações entre eles e entre seus conceitos, juntamente com a cobertura de cada um.

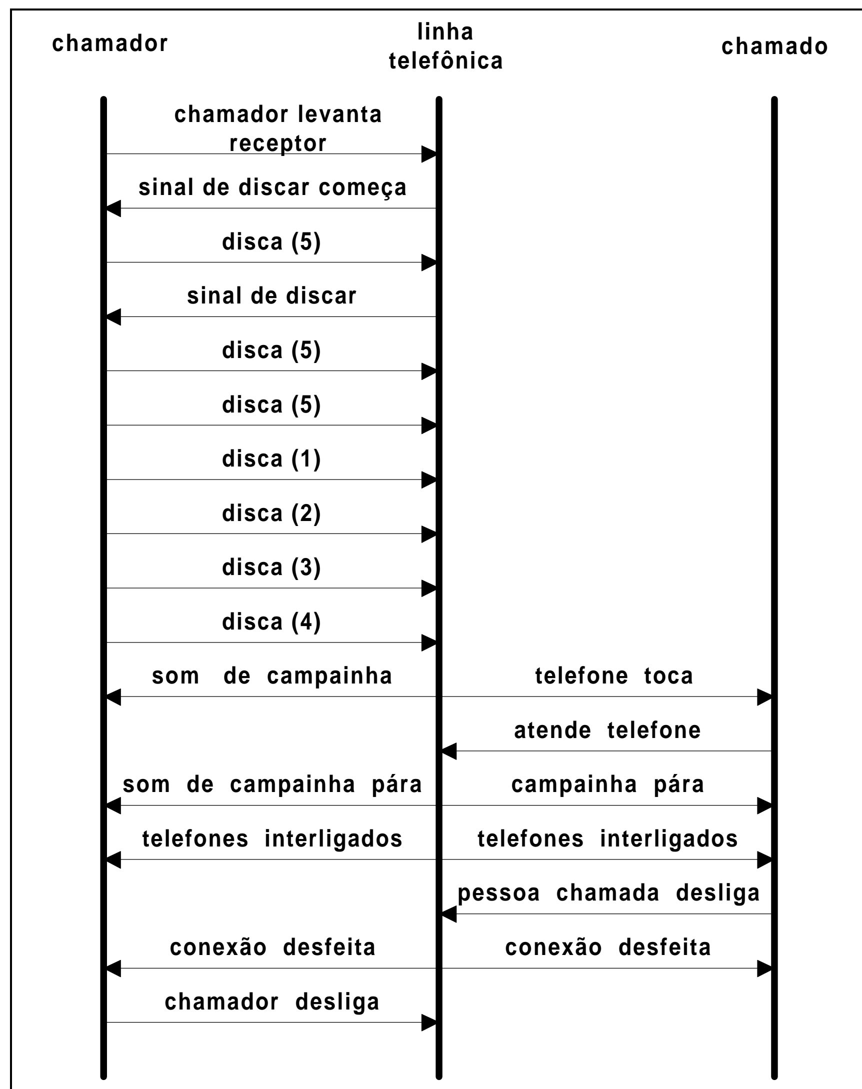

Figura 13 - Um diagrama de eventos para uma chamada telefônica (RUM94)

7. Métodos Orientados a Objeto

Nesta seção serão apresentados cinco métodos orientados a objeto: OOSE, OMT, OOAD, OOA-OOD e FUSION; uma comparação entre os métodos apresentados e seus principais conceitos, pois eles diferem uns dos outros; e, posteriormente, a cobertura de cada um dos métodos.

Antes de ser iniciada a descrição dos métodos faz-se necessária uma explanação sobre a escolha dos métodos.

7.1 Escolha dos métodos

Inicialmente foram selecionados os métodos orientados a objetos mais divulgados e citados em monografias e trabalhos comparativos, como exemplo: [SIL96], [AND95], [BRI95] e [MAR], e que são listados na 1ª coluna da tabela 1.

Na 1ª linha da tabela 1, temos as características:

i) cobertura do método, identificar quais fases do ciclo de desenvolvimento (análise de requisitos, análise, projeto, implementação, teste e manutenção) que cada método abordava; pois seria de interesse para o estudo os que abordassem mais fases; como não é possível estudar todos os métodos, seria escolhido os que cobrissem principalmente análise e projeto;

ii) ferramenta CASE genérica, seis ferramentas CASE mais divulgadas, foram analisadas no sentido de qual método ela fornecia suporte; isto significa que o método é bem divulgado e aceito como eficiente. As ferramentas CASE se encontram na tabela 2;

iii) foi verificado se o método faz parte de algum método integrador, assim verifica-se que o método é aceito por outro metodologista, confirmando sua eficiência. Os métodos integradores observados se encontram na tabela 3.

Para cada uma das características foram dado pontos da seguinte maneira:

1. cobertura do método: 1 ponto para cada fase, (Rq+ , 0,5);
2. ferramenta CASE genérica: 1 ponto para cada ferramenta que abordasse o método;
3. parte de métodos integradores: 1 ponto para cada método integrador que o método faz parte.

Com o total apresentado na última coluna da tabela 1, pode-se perceber que os métodos mais pontuados são:

Object Modelling Technique - James Rumbaugh et. al.

Object Oriented Analysis and Design - Grady Booch

Object-Oriented Analysis - Object-Oriented Design - Peter Coad & Edward Yourdon

Object Oriented Software Engineering - Ivar Jacobson et. al.

Fusion **-** Derek Coleman et.al.

E o método Fusion, que é de 2ª geração também foi escolhido, por ser um método já divulgado e bem estabelecido, diferentemente ao método UM (*Unified Method* - Jacobson, Rumbaugh e Booch), que ainda se encontra em fase de estabelecimento.

Tabela 1 - Maturidade dos métodos básicos orientados a objetos

Método **Autor(es)**	**Cobertura do método**	**Ferramenta genérica**	**Parte de métodos integradores**	**Total**
Designing Object Oriented Software R. Wirfs-Brock et.al.	Rq. P. I. T. M.	-	-	5
Entity Relationship Object Oriented Specification Research Group	A. P. I.	-	-	3
Methodology for Object-Oriented Software Engineering of Systems B. Henderson-Sellers & J. M. Edwards	A. P. I.	-	-	3
Object Modelling Technique J. Rumbaugh et. al.	Rq+. A. P. I.	1 - 2 - 3 - 4 - 5 - 6	4	13,5
Object Oriented Analysis and Design G. Booch	Ri. Rq. A. P. I. M.	1 - 2 - 3	3	12
Object Oriented Analysis and Design J. Martin & J. Odell	A. P.	-	-	2
Object-Oriented Analysis - Object-Oriented Design P. Coad & E. Yourdon	Rq+. A. P. I.	1 - 3 - 5 - 6	-	7,5
Object Oriented Information Engineering J. Martin & J. Odell	A. P. I.	1	-	4
Object Oriented System Analysis S. Shlaer & S. Mellor	A.	1 - 3 - 4	-	4
Object-Oriented System Development D. Champeaux et. al.	Rq+. A. P. I.	-	-	3,5
Object Oriented Software Engineering I. Jacobson et. al.	Rq. A. P. I. T.	1 - 3 - 6	2	10

A tabela 2, como já foi dito, apresenta as seis ferramentas CASE mais amplamente divulgadas: em revistas e Internet.

Na primeira coluna desta tabela temos o nome das ferramentas CASE, na segunda coluna se encontram os fabricantes de cada uma das ferramentas, seguido pelo endereço da Internet, onde podem ser encontradas informações sobre a ferramenta.

Tabela 2 - Ferramentas CASE que suportam métodos orientados a objetos

Ferramenta CASE	Nome do fabricante	Endereço Internet
Paradigm Plus	Platinum Technology	http://www.platinum.com
Rational Rose	Rational Software Corporation	http://www.rational.com
System Architect	Popkin Software & Systems, Inc.	http://www.popkin.com
ObjectTeam	Cayenne Software, Inc.	http://www.cayennesoft.com
Together	Object International, Inc.	http://www.oi.com
Select Perspective	SELECT Software Tools	http://www.pmp.co.uk

A tabela 3 apresenta os métodos integradores, com foi dito anteriormente. A primeira coluna desta tabela mostra o nome do método integrador e seu(s) autor(es); e na segunda coluna, os métodos utilizados para compor o método integrador. Por exemplo: O método FUSION aborda os métodos OMT, OOD, métodos formais e a técnica CRC.

Tabela 3 - Métodos integradores

Método integrador Autor(es)	Métodos utilizados
Catalysis D. D'Souza & A. Wills	OMT (Rumbaugh) / Objectory (Jacobson) Fusion (Coleman)
Fusion D. Coleman et.al.	OMT (Rumbaugh) / OOD (Booch) Técnica CRC (Wirfs-Brock) / Métodos Formais
Syntropy Syntropy User Group	OMT (Rumbaugh) / OOAD (Booch)
Unified Method I. Jacobson - J. Rumbaugh - G. Booch	OMT (Rumbaugh) / OOAD (Booch) Objectory (Jacobson)

7.2 Método *OOSE*

O método *OOSE* (*Object-Oriented Software Engineering*) é uma versão simplificada do método orientado a objetos *Objectory* (*The Object Factory for Software Development*), e foi desenvolvido por Ivar Jacobson e outros, sendo apresentado em [JAC92].

Segundo [JAC92], a complexidade de um sistema deve ser manuseada de uma forma organizada. Isto é feito quando se trabalha com diferentes modelos, cada um focalizando um certo aspecto do sistema. Introduzindo a complexidade gradualmente em uma ordem específica nos modelos sucessivos, a complexidade do sistema será gerenciada.

Este método é derivado de três técnicas totalmente diferentes as quais tem sido usadas por um longo tempo. São elas: programação orientada a objetos, modelagem conceitual e projeto de bloco.

Da programação orientada a objetos, foram pegos os conceitos de encapsulamento, herança e o relacionamento entre classes e instâncias. O objetivo da modelagem conceitual é criar modelos do sistema ou organização a ser analisada. Dependendo do sistema e dos aspectos que se deseja modelar, modelos conceituais diferentes são criados. O projeto de bloco é um método que inicialmente era aplicado a projetos de *hardware* e, agora, também modela *software*, coletando programas e dados juntos em módulos (blocos) e descrevendo suas comunicações mútuas através de interfaces.

Existem três processos principais no método *OOSE*, ilustrado na figura 14: (1) análise, onde se cria uma figura conceitual do sistema que se deseja construir; (2) construção, onde se inclui a implementação e resultados em um sistema completo; e (3) teste, onde se integra o sistema, verifica-o e decide se ele deve ser passado para distribuição.

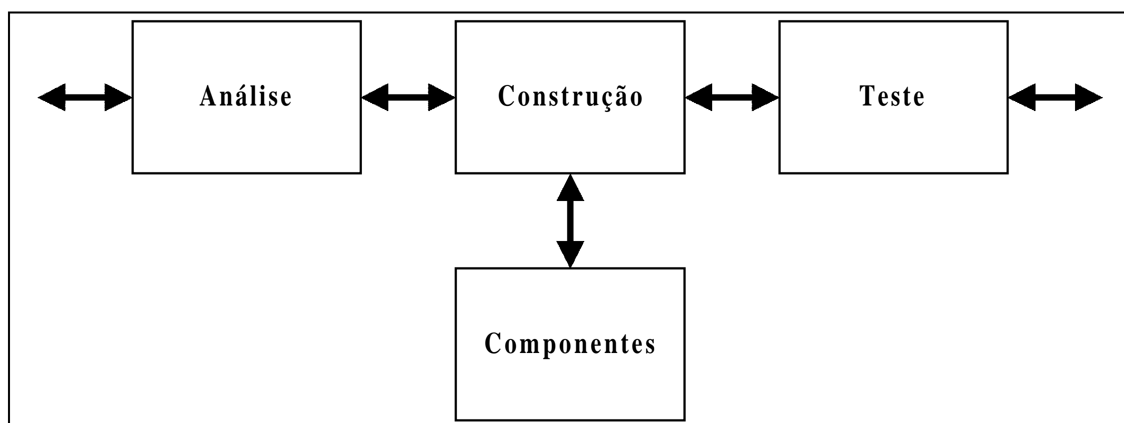

Figura 14 - Os processos do método *OOSE* (JAC92)

Além desses existe o processo de desenvolvimento de componentes, o qual comunica principalmente com o processo de construção. Este processo desenvolve e mantém componentes para serem usados durante a construção. Um componente é uma abstração implementada que é geral e de alta qualidade; ela é desenvolvida e empacotada com o objetivo de reuso.

Os componentes são códigos implementados, os quais podem ser usados em aplicações diferentes. Um componente pode ser uma *white-box*[10] ou uma *black-box*[11].

O processo da análise produz dois modelos, ilustrado na figura 15, o modelo de requisitos e o modelo de análise.

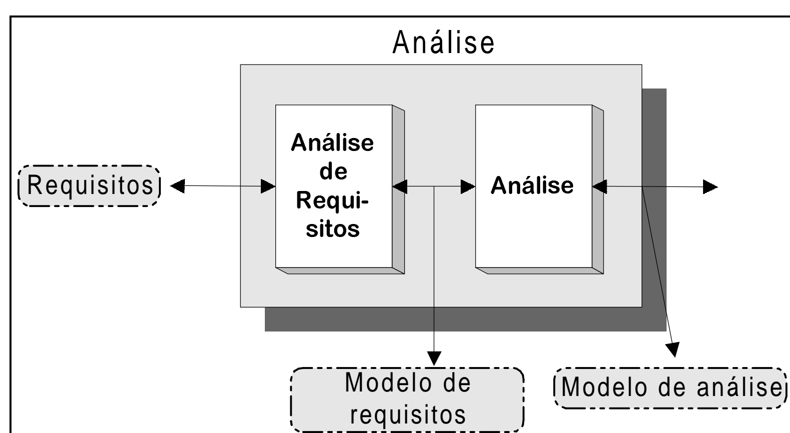

Figura 15 - A fase de análise do método *OOSE* (JAC92)

A partir dos requisitos do sistema, um modelo de requisitos é criado, onde é especificado toda funcionalidade do sistema e definidas as limitações do sistema. Para tanto são desenvolvidos os seguintes modelos: (1) uma ilustração conceitual do sistema, usando os objetos do domínio do problema; (2) a descrição da interface do sistema, se for significativo para o sistema; e (3) um modelo use case, que captura a funcionalidade do sistema. O modelo *use case* também formará a base dos processos de construção e teste, e ele controla uma grande parte do desenvolvimento do sistema.

O modelo de análise é a base da estrutura do sistema. Neste modelo, são especificados todos os objetos lógicos a serem incluídos no sistema e como estes estão relacionados e agrupados. Seu objetivo é estruturar o sistema independentemente do ambiente de implementação atual. A estrutura definida é estável, robusta, manutenível e extensível. Na análise capturamos: (1) a informação contida no sistema; (2) o procedimento que o sistema

[10] *White-box* (caixa branca) - significa que o componente pode ser modificado para reusá-lo. Um tipo especial é uma estrutura, a qual é um esqueleto maior de um projeto que é reusável [JAC92].
[11] *Black-box* (caixa preta) - significa que não podem ser feitas mudanças dentro do componente [JAC92].

adotará; e (3) a representação que provê os detalhes que representam o sistema para o mundo de fora.

O processo de construção (ilustrado na figura 16), projeta e implementa o sistema. A fase de construção necessita como entrada, do modelo de requisitos e do modelo de análise. Primeiro, um projeto é feito resultando em um modelo do projeto, onde cada objeto será totalmente especificado. O subprocesso implementação implementará estes objetos e resultará no modelo de implementação, o qual consiste de código fonte.

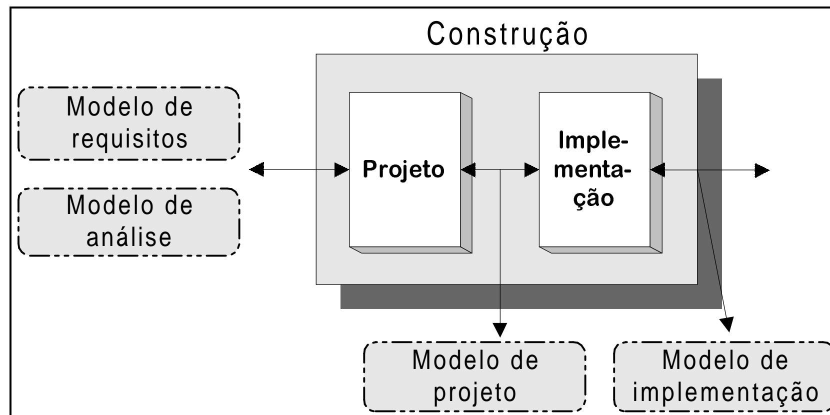

Figura 16 - A fase de construção do método *OOSE* (JAC92)

Através do subprocesso projeto, é criado o modelo de projeto que é um refinamento e formalização do modelo de análise[12]. O primeiro trabalho é adaptar o ambiente de implementação atual. O modelo de análise foi desenvolvido assumindo condições ideais, agora deve-se adaptá-lo a realidade.

O modelo de implementação consiste do código fonte comentado, gerado a partir do subprocesso implementação. A técnica pode ser usada com qualquer linguagem de programação, entretanto, uma linguagem de programação orientada a objetos é desejável, já que todos os conceitos fundamentais podem ser facilmente traçados.

O processo de teste necessita do modelo de requisitos, do modelo de projeto e do modelo de implementação, gerando o modelo de teste. Este processo é ilustrado na figura 17.

Este método, no processo de teste, se preocupa com a verificação, pois segundo [JAC92], a validação é capturada através da análise de requisitos, interações com o cliente e uso de protótipos.

[12] A transição do modelo de análise para o modelo de projeto deve ser feito quando as conseqüências do ambiente de implementação começarem a aparecer [JAC92].

O processo de teste pode ser iniciado junto com a análise e não necessariamente depois da análise e construção; quanto mais cedo for iniciado, menores serão os custos de correção.

O teste de unidade, é o de mais baixo nível e é normalmente feito pelo próprio desenvolvedor, principalmente devido aos custos. Este é, freqüentemente, um teste de *procedures* e subrotinas. Este subprocesso consiste em teste de estrutura, teste de especificação e teste baseado em estado [JAC92].

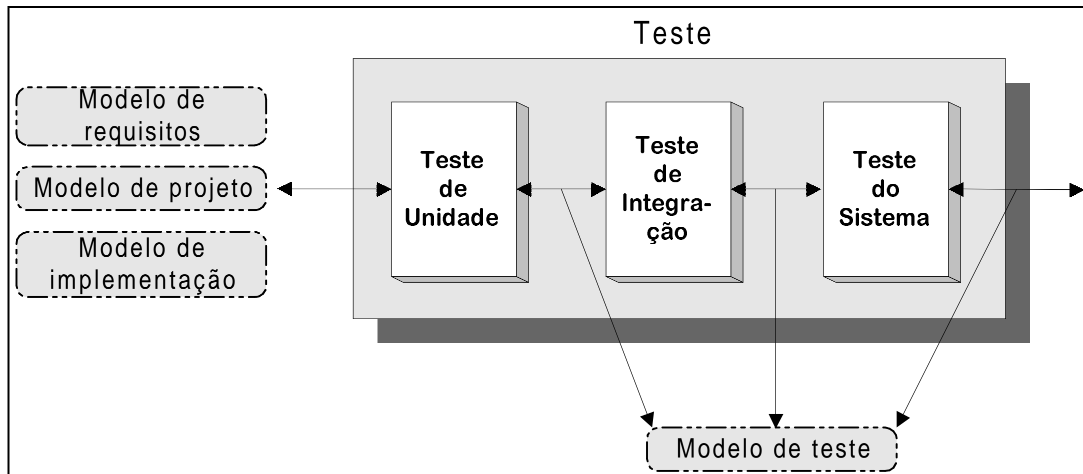

Figura 17 - A fase de teste do método *OOSE* (JAC92)

O subprocesso teste de integração testará se diferentes unidades estão trabalhando juntas com sucesso. Neste processo é incluído o teste de blocos, pacotes de serviço, *use cases*, subsistemas e o sistema todo. Irão existir diversos testes de integração durante o desenvolvimento em diferentes níveis.

No subprocesso teste do sistema, cada *use case* é testado separadamente a partir de uma visão externa; este é baseado no modelo de requisitos. Em seguida, o sistema é testado como um todo.

7.2.1 Considerações finais sobre o método OOSE

Este é o método mais completo, dentre os aqui apresentados, cobrindo todas as fases do processo de desenvolvimento, mas seu enfoque maior é na fase de análise de requisitos.

Apresenta características favoráveis a sistemas que necessitem da participação de usuários e especialistas do domínio no detalhamento dos requisitos, devido a utilização de *use cases*, atores e diagramas de interação.

Os *use cases* conduzem todo o desenvolvimento do sistema, o que permite uma coerência entre os modelos gerados nas diversas fases, pois todos estarão centrados em refinar os *use cases*. Os outros métodos não têm esta abordagem, tentam identificar todos os objetos, suas relações e posteriormente fazer algum tipo de modularização.

O método introduz no modelo de análise objetos extra-domínio (objetos de controle e objetos de interface). A utilização destes tipos de objetos é defendida pelos autores, como sendo uma maneira de facilitar futuras evoluções e manutenções no sistema, pois estas ficariam mais localizadas, afetando apenas os objetos com maior chance de modificações (interface e controle).

A técnica de análise de OOSE é a evolutiva orientado a dados, que utiliza extensões semânticas de modelos de dados [REI94].

A notação que o método utiliza é bastante diferente de todos os outros métodos aqui apresentados, os quais têm algumas características comuns entre si. Por exemplo, a representação dos atributos externa a representação do objeto, o que dificulta a compreensão dos diagramas para os iniciantes.

O método pode ser utilizado no desenvolvimento de sistemas reais de grande porte. É indicado para ser usado por equipes de desenvolvimento e não por um único desenvolvedor; sendo que as esquipes devem ter treinamento em análise, projeto e programação.

7.3 Método OMT

Do inglês *Object Modeling Technique*; também conhecido como TMO, Técnica de Modelagem de Objetos, OMT é um método orientado a objetos desenvolvido por James Rumbaugh e outros, sendo apresentado em [RUM94].

A essência do desenvolvimento orientado a objeto no OMT é a identificação e organização de conceitos do domínio da aplicação ao invés de conceitos do domínio da implementação.

A figura 18 apresenta uma visão geral a respeito do processo de desenvolvimento de *software* do método OMT.

O método OMT consiste de 3 fases: análise, projeto e implementação.

Na fase de **análise** encontram-se três modelos: (i) o modelo de objetos, que representa os aspectos estáticos e estruturais "de dados" de um sistema; (ii) o modelo dinâmico, que representa os aspectos temporais e comportamentais "de controle" de um sistema; e (iii) o modelo funcional, que representa os aspectos relativos às transformações "de funções" de um sistema.

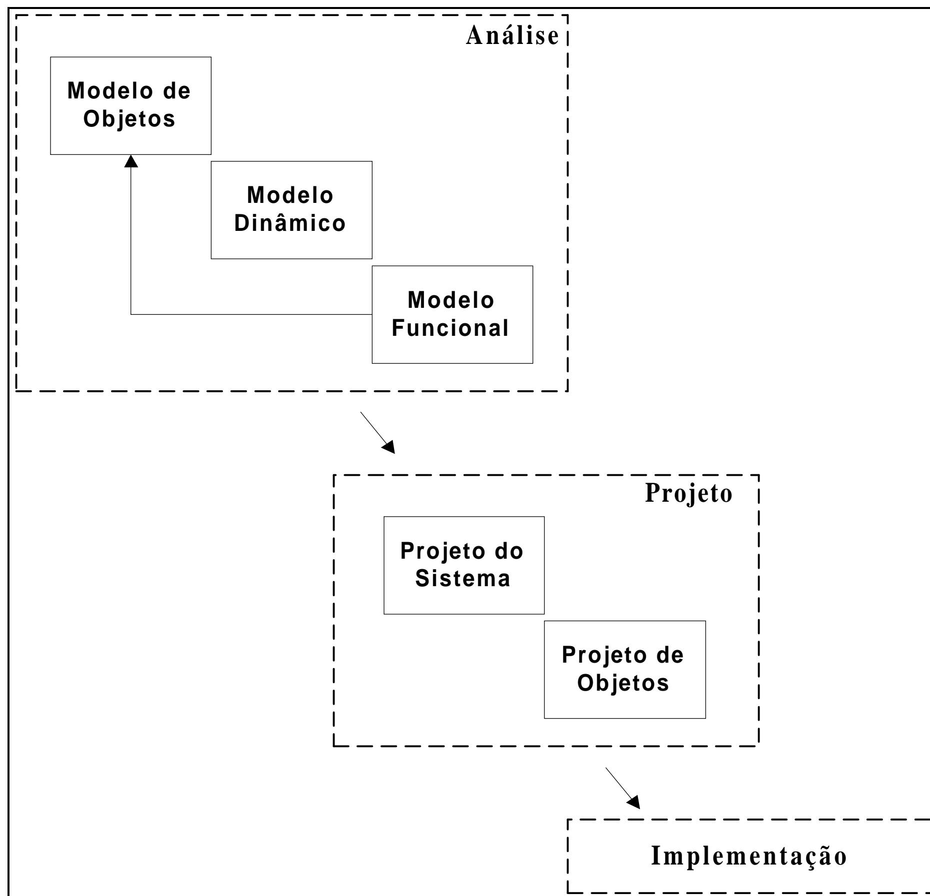

Figura 18 - Método OMT (COL96)

i) **Modelo de Objetos**: descreve a estrutura de objetos de um sistema - sua identidade, seus relacionamentos com outros objetos, seus atributos e suas operações. A meta é incorporar os conceitos do mundo real que sejam importantes para a aplicação. O modelo de objetos é representado graficamente por diagramas de objetos[13] contendo classes de objetos, e é descrito por um dicionário de dados.

ii) **Modelo Dinâmico**: descreve os aspectos de um sistema relacionados ao tempo e à seqüência de operações. Ele incorpora o controle, que é um aspecto de um sistema que descreve as seqüências de operações que ocorrem, independentemente do que as operações fazem, sobre o que elas atuam ou como são implementadas. O modelo é descrito por diagramas de estados e diagramas de fluxo de eventos. [RUM94] sugere a criação dos cenários e, baseando-se nestes, é feita a construção de diagrama de eventos. O diagrama de estados relaciona eventos e estados. Ele especifica a seqüência de estados causados por uma seqüência de eventos para um objeto específico.

iii) **Modelo Funcional**: descreve os aspectos de um sistema relacionados a transformações de valores: funções, mapeamentos, restrições e dependências funcionais. O modelo abrange o que um sistema faz, independentemente de como ou quando é feito. O modelo funcional especifica o significado das operações do modelo de objetos e as ações do modelo dinâmico, bem como quaisquer restrições no modelo de objetos. Ele é representado por múltiplos diagramas de fluxo de dados, que especificam o significado das operações e restrições.

Na figura 18, ilustrada anteriormente, a seta que sai do modelo funcional e retorna ao modelo de objetos indicando que há uma interação para verificar e refinar os modelos.

Na fase de **projeto** encontram-se dois tipos de projetos: (i) o projeto de sistema; e (ii) o projeto de objetos.

i) **Projeto de Sistema**: deve-se tomar as seguintes decisões: organizar o sistema em subsistemas, identificar concorrências inerentes ao problema; alocar subsistemas aos processadores e tarefas; escolher uma abordagem para o gerenciamento de depósitos de dados (depósito de dados internos ou externo ou o uso de um SGBD - Sistema Gerenciador de Banco de Dados); cuidar dos acessos aos recursos globais[14]; escolher a implementação de controle em *software* (controle externo ou interno); tratar das condições limites (inicialização, término e falhas) e ajustar o equilíbrio das prioridades (uso de mais memória ou da prototipação).

ii) **Projeto de Objetos**: deve-se executar as seguintes etapas: combinar os três modelos da análise para obter as operações sobre classes; projetar algoritmos para implementar operações; otimizar caminhos de acesso aos dados; implementar controle para interações externas (implementar o modelo dinâmico); ajustar a estrutura de classes para aumentar a herança; projetar associações (implementar todas as associações de maneira uniforme ou selecionar uma técnica especial para cada associação); determinar a representação de objetos (tipos primitivos ou combinar grupos de objetos relacionados) e empacotar classes e associações em módulos.

A fase de **implementação**: as classes de objetos e os relacionamentos desenvolvidos durante o projeto de objetos são por fim traduzidos para uma determinada implementação em

[13] Existem dois tipos de diagramas de objetos: 1) diagrama de classes, que descreve muitas instâncias possíveis de dados e também classes de objetos; e 2) diagrama de instâncias, que descrevem como os objetos de um determinado conjunto se relacionam entre si [RUM94].

[14] Recursos globais incluem: unidades físicas, espaço, nomes lógicos e acesso a dados compartilhados [RUM94].

uma linguagem de programação, em um banco de dados ou em *hardware*. A programação deve ser uma parte mecânica e relativamente de menor importância do ciclo de desenvolvimento, porque todas as decisões difíceis devem ser tomadas durante o projeto.

7.3.1 Considerações finais sobre o método OMT

Neste método os conceitos de orientação a objetos são bem explicados e exemplificados.

Na fase de análise, o diferencial em comparação com outros métodos está na representação das relações entre classes e objetos, com características não encontradas em outros métodos, como: relações ternárias, atributos de ligação e qualificador de ligação. Sua notação para os modelos de análise é a mais clara dentre as aqui apresentadas, facilitando a visualização das estruturas.

A fase de análise do OMT leva a identificação de objetos do domínio como os principais objetos do sistema. Esta abordagem é diferente da adotada por alguns métodos (como OOSE), que defendem a utilização de objetos extra-domínio nesta fase com o objetivo de gerar uma arquitetura mais fácil de evoluir. OMT defende a utilização de objetos do domínio por serem mais facilmente compreendidos pelos especialistas no domínio, os quais não são especialistas em modelagem de sistemas.

A técnica de análise do OMT é a integracionista, que representa aquelas técnicas que integram modelos separados das diferentes dimensões (no caso deste método as dimensões são: estática, dinâmica e funcional), não necessariamente de todas estas três (anteriormente citadas) [REI94].

A fase de analise é bastante prescritiva, facilitando o desenvolvimento das atividades. A fase de projeto não apresenta o mesmo detalhamento e riqueza semântica das representações utilizadas na análise, dando apenas sugestões genéricas de como evoluir os modelos de análise para os de projeto.

Para o detalhamento das operações das classes, OMT utiliza os Diagramas de Fluxo de Dados (DFDs). Os DFDs são interessantes quando tivermos um fluxo de dados sofrendo transformações ao longo das operações, mas não são interessantes para representar fluxo de controle, que é melhor representado por fluxogramas como em OOA (Coad/Yourdon).

O método é uma boa opção para analistas experientes em modelagem com métodos estruturados de análise (sem muita experiência em programação) pois o seu enfoque é na abstração no domínio do problema e não na implementação. Facilita o trabalho dos analistas com várias opções de representação das associações entre objetos.

O método não é uma boa opção para programadores que estejam aprendendo a modelar, pois o seu enfoque não é direcionado a linguagens de implementação, mas sim a abstrações de alto nível. Os programadores tendem a querer mapear as representações de análise para algo no domínio de implementação, mas os modelos de análise não têm este objetivo, gerando alguma confusão nos primeiros projetos.

Em resumo é um método com um forte enfoque conceitual, uma fase de análise bastante clara, mas incompleto na fase de projeto. Pode ser utilizado no desenvolvimento de sistemas reais por desenvolvedores experientes (que saberão conduzir a fase de projeto sem as prescrições que o método não fornece) ou com o complemento de outros métodos que sejam mais abrangentes na fase de projeto com OOAD (Booch) ou OOSE (Jacobson).

7.4 Método OOAD

O método OOAD - *Object-Oriented Analysis and Design* - Análise e Projeto Orientado a Objeto, é um método orientado a objetos desenvolvido por Grady Booch. O método foi inicialmente criado somente com a fase de projeto, o OOD (*Object-Oriented Design*), sendo apresentado em [BOO91]. Em [BOO94], Booch incorpora ao seu método a fase de análise, o OOAD.

O método OOAD é dividido em (i) macro processo e (ii) micro processo, que serão descritos logo a seguir.

> *"Use o macro processo para controlar as atividades do projeto como um todo, e use o micro processo para interativamente executar estas atividades e regular a futura conduta do macro processo".* ***Booch***

i) **<u>Macro processo</u>**

O macro processo serve como uma estrutura de controle para o micro processo. A principal preocupação do macro processo é o gerenciamento técnico da equipe de desenvolvimento.

Como ilustrado na figura 19, o macro processo apresenta as seguintes atividades: (1) conceitualização; (2) análise; (3) projeto; (4) evolução; e (5) manutenção, que serão descritas a seguir.

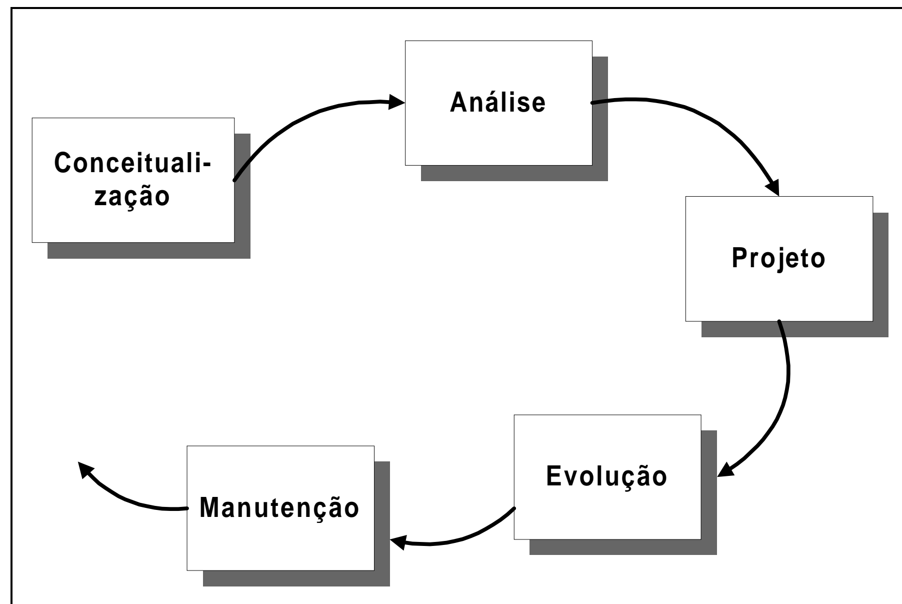

Figura 19 - Macro processo do método OOAD (BOO94)

1) Conceitualização

O propósito da conceitualização é estabelecer o núcleo dos requisitos para o sistema. Para uma nova parte do sistema ou para adaptação de um sistema existente. Nem todos os projetos requerem esta fase.

Durante esta fase, três produtos são gerados. Em ordem de importância: (1) um protótipo executável; (2) uma avaliação dos riscos; e (3) uma visão geral dos requisitos do projeto.

> *"Faça a conceitualização somente quando os riscos do projeto forem relativamente altos ou quando é necessário inventar um vínculo inicial entre o cliente e a organização de desenvolvimento; senão, passe diretamente para a análise."* ***Booch***

2) Análise

O propósito da análise é desenvolver um modelo do comportamento do sistema. Este modelo identifica classes e objetos (suas regras, responsabilidades e colaborações) que formam o vocabulário do domínio do problema.

Durante a análise, quatro produtos são gerados, todos com importância quase igual:

(1) uma descrição do contexto do sistema, que estabelece os limites do projeto e faz uma distinção entre os elementos que estão de dentro e de fora do sistema;

(2) uma coleção de cenários que definem o comportamento do sistema;

(3) um modelo do domínio, o propósito é visualizar todas as classes centrais responsáveis pelo comportamento essencial do sistema;

(4) uma revisão da avaliação dos riscos, o produto final da análise é uma avaliação dos riscos que identifica a área de conhecimento dos riscos técnicos e não-técnicos que podem ter impacto no projeto. Se existir a fase de conceitualização, então esta avaliação dos riscos é justamente uma evolução daquele, adicionando o que foi aprendido na análise.

3) Projeto

O propósito do projeto é criar uma arquitetura para desenvolver a implementação. O projeto arquitetural só pode começar quando a equipe de desenvolvimento tem um entendimento razoável dos requisitos do sistema. O projeto focaliza a estrutura, tanto estática como dinâmica. Análise mais aprofundada pode ocorrer durante a fase do projeto, principalmente para explorar áreas de incerteza a respeito do comportamento desejado do sistema, mas o projeto principalmente serve para criar o esqueleto concreto sobre o qual todo o resto da implementação se projeta.

Durante o projeto, são gerados cinco produtos:

(1) um executável e uma arquitetura de linha básica[15];

(2) a especificação de todos modelos arquiteturais importantes;

(3) um planejamento do *release*[16];

(4) critérios de teste; e

(5) uma revisão da avaliação de riscos.

Os dois primeiros estabelecem a estrutura do sistema e os outros três suportam o processo de desenvolvimento baseado em risco para desenvolver a estrutura. Uma arquitetura executável[17] é o mais importante produto desta fase de desenvolvimento.

---

[15] Do inglês *'baseline'*, linha básica é um *software/hardware* que foi formalmente revisado e combinado, o qual serve como a base para novo desenvolvimento, e que pode ser mudado somente através de procedimentos de controle de mudança formal. É um documento ou conjunto de documentos de identificação da configuração software/hardware formalmente revisados e combinados no tempo específico durante o ciclo de vida do sistema (*software*), o qual descreve completamente as características físicas e/ou funcionais de um item de configuração de *hardware/software* [THA88].

[16] Um *release* é simplesmente uma versão executável, completa e estável de um sistema, junto com quaisquer outros elementos periféricos necessários para usá-lo. Os produtos da evolução guiam a equipe de desenvolvimento através da conclusão de uma solução que satisfaça os requisitos reais do cliente. Um *release* consiste de um pacote cheio, incluindo *software* executável, documentação e resultados de teste [BOO96].

[17] Uma arquitetura executável existe como aplicação real que executa de maneira limitada; é a produção de um código de qualidade; leva alguns ou todos os procedimentos de um pouco de cenários interessantes escolhidos a partir da fase de análise [BOO96].

4) Evolução

O propósito da evolução é produzir uma implementação através de sucessivos refinamentos da arquitetura do sistema. O fim desta fase é disponibilizar o sistema, isto envolve uma completa produção do sistema, seu *software* (executável), mecanismos de instalação, documentação e facilidades de suporte.

A evolução de um sistema produz quatro produtos distintos: (1) uma torrente de *releases* executáveis; (2) protótipos comportamentais; (3) resultados com garantia de qualidade; e (4) documentação do usuário e do sistema.

5) Manutenção

O propósito da manutenção é gerenciar o sistema gerado pela evolução. Esta fase é largamente a continuação da anterior. De fato, mudanças mais localizadas são feitas ao sistema como adicionar alguns requisitos novos e aniquilar erros protelados.

Desde que a manutenção está no sentido de continuar a evolução do sistema, seus produtos são idênticos aqueles da fase anterior, com uma adição: uma lista (*punch list*) de novas tarefas. Esta lista serve como veículo para coletar erros e aumentar requisitos, tanto que eles podem ser priorizados para manutenção futura de *releases*.

***ii)* Micro processo**

O micro processo é a descrição das atividades diárias de um único ou pequeno grupo de desenvolvedores de *software*. O micro processo pode, a partir de um observador externo, parecer obscurecido porque as fases de análise e projeto não são claramente definidas.

A figura 20 ilustra o micro processo do método OOAD, o qual contém quatro atividades: (1) identificação de classes e objetos; (2) identificação da semântica das classes e objetos; (3) identificação dos relacionamentos entre classes e objetos; e (4) implementação de classes e objetos; que serão explicadas logo a seguir.

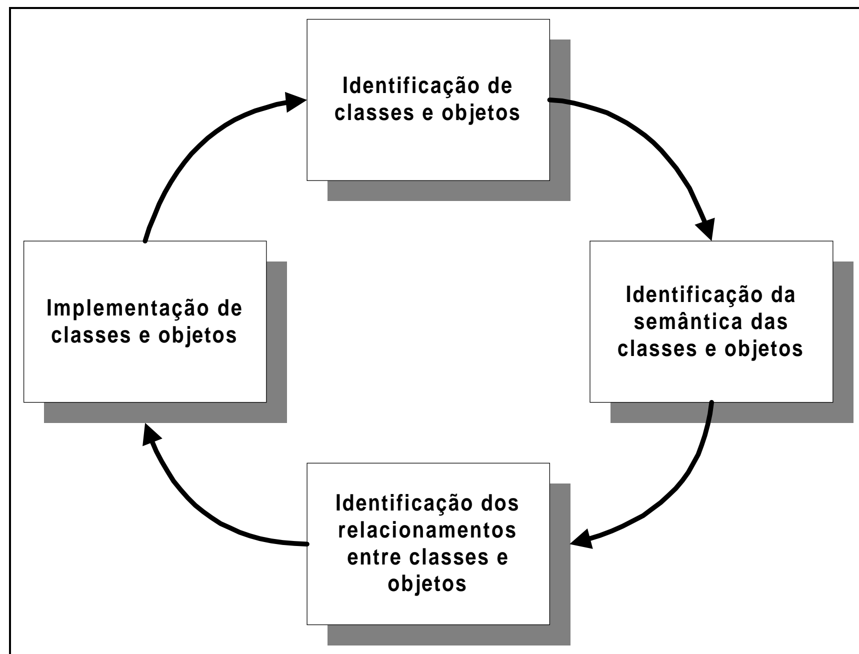

Figura 20 - Micro processo do método OOAD (BOO94)

1) Identificação de classes e objetos

O propósito desta fase é estabelecer os limites do problema. Esta é uma fase de descobrimento. Durante a análise, este passo é aplicado para descobrir as abstrações que formam o vocabulário do domínio do problema, assim, a equipe de desenvolvimento começa a restringir o problema, decidindo o que é e o que não é de interesse. Durante o projeto, este passo serve para descobrir abstrações que são elementos da solução. Durante a evolução, esta fase identifica as abstrações de mais baixo nível que constróem as de mais alto nível e pela identificação de elementos comuns entre as abstrações existentes para simplificar a arquitetura do sistema.

Durante esta atividade, existe um único produto, o dicionário de dados, que serve como um repositório central para as abstrações relevantes do problema.

2) Identificação da semântica das classes e objetos

O propósito é estabelecer os procedimentos e atributos de cada abstração identificadas anteriormente e fazer a distribuição adequada, refinando as abstrações candidatas.

Durante análise, esta atividade é aplicada para alocar as responsabilidades a diferentes procedimentos do sistema. Durante o projeto, este passo serve para estabelecer uma separação clara de assuntos entre as partes da solução. Durante a evolução, esta fase visa gradualmente e conscientemente refinar estas semânticas, transformando-as a partir de descrições de forma livre em protocolos concretos para cada abstração.

3) Identificação dos relacionamentos entre classes e objetos

O propósito desta atividade é solidificar os limites das abstrações e reconhecer os colaboradores de cada abstração já identificada nas fases anteriores.

Durante a análise, este passo primariamente especifica associações entre classes e objetos (incluindo certos relacionamentos importantes de herança e agregação). Entre outras coisas, estas associações especificam alguma dependência semântica entre duas abstrações e alguma habilidade de navegar de uma entidade para outra. Durante o projeto, serve para especificar as colaborações que formam os mecanismos de nossa arquitetura e agrupar as classes de mais alto nível em categorias e módulos dentro de subsistemas. Durante a evolução, esta fase continua a adicionar detalhes a arquitetura do sistema, com o refinamento dos relacionamentos.

4) Implementação de classes e objetos

O propósito desta fase é representar cada abstração e mecanismos concretamente de maneira mais eficiente e elegante.

Durante análise, o projeto bem sucedido procederá com uma modesta implementação, primariamente como um recurso para afastar o risco e como um veículo para revelar novas classes e objetos para serem refinados na próxima interação. Durante o projeto, esta fase serve para criar produtos tangíveis e executáveis pelos quais a arquitetura do sistema toma forma. Durante a evolução, a atividade nesta fase do micro processo acelera, tendo um sucessivo refinamento da arquitetura.

A atividade associada a este passo é a seleção das estruturas e algoritmos que completam as regras e responsabilidades de todas as várias abstrações identificadas mais cedo no micro processo. Enquanto que as primeiras três fases do micro processo focalizam a visão externa das abstrações, este passo focaliza a visão interna.

Tipicamente, a análise incluirá conjuntos de diagramas de objetos, diagramas de classe e diagramas de transição de estado. O projeto (incluindo arquitetura e implementação) incluirá conjuntos de diagramas de classes, diagramas de objetos, diagramas de módulos e diagramas de processo, tão bem como sua visão dinâmica correspondente, diagrama de interação.

Apresenta-se uma breve descrição dos diagramas citados anteriormente:

i) Diagrama de classes - é usado para mostrar existência de classes e seus relacionamentos no projeto lógico de um sistema. Um simples diagrama de classes representa uma visão da estrutura de classe de um sistema;

ii) Diagrama de objetos - é usado para mostrar existência de objetos e seus relacionamentos no projeto lógico de um sistema. Um simples diagrama de objetos é tipicamente usado para representar um cenário;

iii) Diagrama de interação - é usado para traçar a execução de um cenário no mesmo contexto de um diagrama de objeto;

iv) Diagrama de transição de estado - é usado para mostrar o estado de uma instância de uma dada classe, os eventos que causam a transição de um estado para outro e as ações que resultam da mudança de estado;

v) Diagrama de módulo - é usado para mostrar a alocação de classes e objetos em módulos no projeto físico de um sistema. Um simples diagrama de módulo representa uma visão da arquitetura de módulos de um sistema;

vi) Diagrama de processo - é usado para mostrar alocação de processos a processadores no projeto físico de um sistema. Um simples diagrama de processo representa uma visão da arquitetura de processo de um sistema.

7.4.1 Considerações finais sobre o método OOAD

O método utiliza o macro processo para controlar as atividades do desenvolvimento, e o micro processo, para executar estas atividades.

A técnica de análise do OOAD é a reversa, que é originada da necessidade de implementação, como, por exemplo, o suporte a conceitos de linguagens de programação orientadas a objetos específicas; no caso deste método, se originou da linguagem ADA [REI94].

A parte conceitual sobre classes e objetos é bastante didática, expondo de maneira precisa vários conceitos relacionados a objetos.

Não é um método prescritivo, como o OOA-OOD (Coad/Yourdon) e o OMT (Rumbaugh), sendo portanto difícil a sua aplicação por equipes inexperientes. É um método bom para equipes de programadores experientes (em linguagens O.O.) que julguem não ser necessária a elaboração de modelos de requisitos e análise e visualizam o projeto de uma forma interativa com a implementação (documentar a implementação).

Não é um bom método quando se deseja gerar modelos de requisitos e análise. Em OOAD, Booch sugere a utilização de conceitos de outros métodos (OOSE e OMT) na sua fase de análise, tornando bastante confusa a notação e os conceitos que devem ser utilizados.

7.5 Método OOA-OOD

OOA-OOD é um método orientado a objetos desenvolvido por Peter Coad e Edward Yourdon que trata as fases de análise, projeto e programação. O método é apresentado em [COA91], cobrindo a parte de análise e em [COA93], cobrindo a parte de projeto e programação.

A análise (OOA) identifica e define classe&objetos[18] que refletem diretamente o domínio do problema e as responsabilidades do sistema dentro dele.

O projeto (OOD) identifica e define classe&objetos adicionais, refletindo a implementação dos requisitos (nível de diálogo, nível de administração de tarefas, nível de administração de dados).

A fase de análise é composta por cinco atividades: (1) encontrar classe&objetos; (2) identificar estruturas; (3) identificar assuntos[19]; (4) definir atributos; e (5) definir serviços. Estas atividades não são passos seqüenciais, ficando a critério do analista a ordem a ser seguida, além disso, há uma interatividade entre elas.

1) Encontrar classe&objetos - para que isso ocorra de maneira bem sucedida, [COA91] sugere: (1) observar o domínio da aplicação; (2) ouvir os especialistas do domínio da aplicação; (3) checar resultados de análise feitas anteriormente para domínio de aplicação similar, verificando a possibilidade de reuso; (4) checar outros sistemas do mesmo domínio, verificando o que pode ser aprendido em relação a identificar classe&objetos; (5) assegurar-se dos requisitos existentes em um documento feito pelo cliente; e (6) utilizar a prototipação como meio de obter uma análise efetiva.

Nesta fase deve-se procurar por estruturas, outros sistemas, dispositivos, coisas ou eventos lembrados, identificar papéis executados[20], procedimentos operacionais, lugares e unidades organizacionais.

2) Identificar estruturas - esta atividade é realizada criando-se dois tipos de estruturas: (1) a estrutura de generalização, que captura a hierarquia de herança entre as classes identificadas; e (2) a estrutura de agregação, que modela as partes de um objeto e como os objetos são agrupados em categorias mais amplas.

3) Identificar assuntos - esta fase é realizada para facilitar o entendimento do leitor através de um modelo amplo e complexo. É feita transformando a classe mais superior de

[18] Classe&objetos é um termo significando “uma classe e os objetos nessa classe” [COA91].
[19] Assuntos são grupos de classe&objetos.
[20] Existem dois tipos de papéis executados: (1) usuários do sistema; e (2) pessoas que não interagem diretamente como o sistema, mas de quem a informação é mantida pelo sistema (proprietário de um motor de veículo).

cada estrutura em um assunto. Uma classe&objetos pode ser encontrada em mais de um assunto se necessário.

4) Definição de atributos - consiste em identificar os atributos; aplicar herança em estruturas de generalização, posicionando-os nas classes apropriadas; identificar conexões de ocorrências entre objetos; verificar casos especiais de atributos[21] e de conexões[22] e especificar os atributos (nome, descrição, especificações).

5) Definição de serviços - para definir os serviços das classes, são realizados os seguintes passos: (1) identificar os estados de objeto, os valores para os atributos; (2) identificar os serviços simples: criar, conectar, acessar e liberar, e complexos: calcular e monitorar; (3) identificar a comunicação entre objetos usando conexão de mensagens; (4) especificar os serviços no modelo de classe&objetos usando um diagrama de serviços para cada serviço; e (5) reunir a documentação da análise.

A documentação da análise reúne:

i) o modelo OOA, o qual consiste de cinco camadas: assunto, classe&objetos, estrutura, atributo e serviço, sendo ilustrado na figura 21;

ii) as especificações de classe&objetos[23], geradas na primeira atividade apresentada;

iii) documentação suplementar, na medida do necessário: tabelas de linhas críticas de execução, especificações adicionais do sistema e tabela de serviços/estados.

Figura 21 - O modelo OOA - um modelo multicamada (COA91)

Na fase de projeto é criado um modelo geral de multicamadas e multicomponentes. O modelo OOD, exatamente como o modelo da OOA, consiste em cinco camadas que são cortes horizontais no modelo geral. O modelo também possue quatro componentes: Componente de Domínio de Problemas; Componente de Interação Humana; Componente de Gerenciamento

[21] Casos especiais de atributos: valor não aplicável, classe&objetos com apenas um atributo, valores de repetição.
[22] Casos especiais de conexões: conexão muitos-para-muitos, conexão entre objetos de uma classe única, múltiplas conexões entre objetos.
[23] As especificações de classe&objetos estão definidas no modelo de especificação, o qual inclue o diagrama de estado do objeto e o diagrama de serviço.

de Tarefas e Componente de Gerenciamento de Dados, que são cortes verticais no modelo geral. O modelo OOD é ilustrado na figura 22.

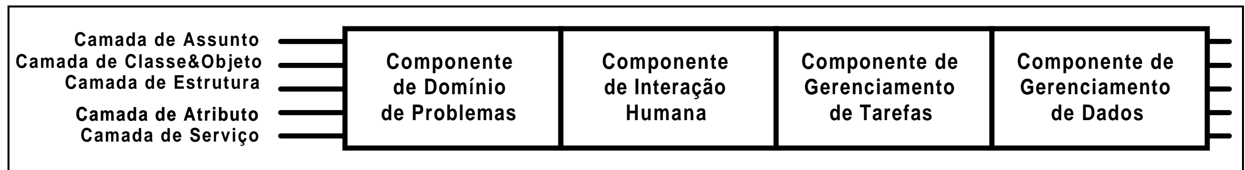

Figura 22 - O modelo OOD - um modelo multicamadas e multicomponentes (COA93)

Componente de Domínio de Problemas (CDP) - os resultados da OOA se encaixam exatamente no CDP. Os resultados da análise são uma parte essencial do modelo multicomponentes do OOD.

Componente de Interação Humana (CIH) - a interação humana precisa de exame detalhado, tanto na análise como no projeto. Na OOA, esse exame é feito de modo que os atributos e os serviços requeridos sejam especificados. No OOD, o CIH acrescenta a esses resultados o projeto da interação humana e os detalhes dessa interação. Esse componente capta como uma pessoa comandará o sistema e como o sistema apresentará as informações ao usuário.

Componente de Gerenciamento de Tarefas[24] (CGT) - Para certas espécies de sistemas são necessárias múltiplas tarefas, por exemplo: (1) para certas interfaces humanas; (2) para sistema multiusuário; (3) para arquiteturas de *software* com muitos subsistemas; (4) para muitas tarefas em um único processador; (5) para arquiteturas de *hardware* com multiprocessador; e (6) para um sistema responsável pela comunicação com outro sistema.

Componente de Gerenciamento de Dados (CGD) - Este componente fornece a infra-estrutura para o armazenamento e a recuperação de objetos de um sistema de gerenciamento de dados. O componente isola o impacto do esquema de gerenciamento de dados, seja ele arquivo simples, relacional ou baseado em objetos.

Os quatro componentes correspondem às quatro atividades principais do OOD: (1) projetar o CDP; (2) projetar o CIH; (3) projetar o CGT; e (4) projetar o CGD; essas atividades como as atividades da análise não são passos seqüenciais.

[24] Tarefa é outro nome para um processo (um fluxo de atividades, definido pelo seu código). A execução concorrente de uma série de tarefas é referenciada como multitarefa.

1) Projetar o CDP - a estratégia utilizada consiste em: (1) aplicar a OOA; (2) aperfeiçoar os resultados da OOA durante o OOD, devido a mudança dos requisitos ou devido a falta de compreensão por parte do analista ou dos especialistas em domínio de problemas disponíveis; e (3) acrescentar algo aos resultados da OOA durante OOD, os critérios de adição são resumidos na figura 23.

Reutilizar as classes de projeto e programação.
Agrupar as classes específicas do domínio do problema.
Estabelecer um protocolo pela adição de uma classe de generalização.
Acomodar o nível de herança suportado.
Melhorar a peformance.
Suportar o Componente Gerenciamento de Dados.
Acrescentar componentes de nível mais baixo.
Não modificar apenas para refletir atribuições de equipe.
Revisar e reivindicar os acréscimos aos resultados da OOA.

Figura 23 - Critérios de adição ao CDP (COA93)

2) Projetar o CIH - a estratégia consiste em: (1) classificar as pessoas; (2) descrever as pessoas e seus cenários de trabalhos; (3) projetar a hierarquia de comando[25]; (4) projetar a interação detalhada; (5) continuar no sentido da prototipação; (6) projetar as classes do CIH; e (7) projetar, levando em conta as Interfaces Gráficas com o Usuário.

3) Projetar o CGT - a estratégia consiste em: (1) identificar tarefas dirigidas por eventos; (2) identificar tarefas dirigidas por tempo; (3) identificar tarefas prioritárias e tarefas críticas; (4) identificar um coordenador; (5) desafiar cada tarefa; e (6) definir cada tarefa.

O ponto principal dessa estratégia é identificar e projetar as tarefas (programas) e os serviços incluídos em cada tarefa.

4) Projetar o CGD - o projeto deste componente precisa incluir: (1) o projeto do *layout* dos dados de arquivos simples, relacional ou baseado em objetos, e (2) o projeto dos serviços correspondentes, acrescentando um atributo e serviço a cada classe&objetos com objetos a serem armazenados.

Sobre a implementação, os autores sugerem o uso de linguagens de programação baseadas em objetos, pois estas são excelentes para suportar as construções da análise e do projeto orientados a objetos. Porém, várias organizações utilizam-se de linguagens não

baseadas em objetos, que podem ser utilizadas, mas que deverão vir combinadas com manuais, disciplina e perseverança, para poderem implementar adequadamente os resultados da análise e do projeto.

[COA93] também avalia as principais linguagens baseadas em objetos: C++, *Object Pascal*, *Smalltalk*, *Objective-C*, *Eiffel*; e não baseadas em objetos: *Ada* e C.

7.5.1 Considerações finais sobre o método OOA-OOD

É o método mais prescritivo dos métodos aqui apresentados, indicando com detalhes todas as atividades que devem ser executadas e os seus subprodutos. O enfoque do método é tanto para a análise quanto para o projeto.

É uma boa opção para equipes iniciantes em tecnologia de objetos pois sua notação é simples e suficiente para boa parte dos projetos orientados a dados, com exceção da representação da cardinalidade, que se encontra do lado oposto, diferentemente do que é representado pelos outros métodos.

É um método fraco na representação da dinâmica do sistema, pois não possui cenários nem diagramas de eventos como em OMT e OOSE.

Durante a explanação do método em [COA91] e [COA92], os autores sugerem o reuso, insistindo que o desenvolvedor faça uma busca em projeto anteriormente desenvolvidos.

7.6 Método FUSION

O FUSION é um método orientado a objetos que incorpora partes bem-sucedidas de métodos anteriores orientados a objetos (OMT, CRC[26], Métodos Formais[27] e OOD), ilustrado na figura 24, procurando resolver seus pontos fracos. Foi desenvolvido por Derek Coleman e outros e sua completa descrição é encontrada em [COL96].

---

[25] Uma hierarquia de comando pode ser: uma série de telas de menu, uma barra de menu ou uma série de ícones que executam ações quando algo é colocado sobre eles.

[26] CRC - Classe-Responsabilidade-Colaborador é uma técnica exploratória, e não um método completo. Foi projetado, originalmente, como uma maneira para ensinar os conceitos básicos de orientação a objetos [COL96].

[27] Os métodos formais representam uma abordagem de engenharia de *software* baseada na aplicação de matemática discreta (ou seja, teoria de conjuntos, lógica, etc.) à programação de computadores. As linguagens de especificação, influenciadas por estes formalismos, permitem que as propriedades essenciais de um sistema sejam capturadas em um alto nível de abstração.

O método possui três fases: (1) análise, que descobre os objetos e as classes existentes no sistema, descreve seus relacionamentos e define as operações que poderão ser executadas pelo sistema; (2) projeto, que decide como representar as operações do sistema através de interações entre objetos relacionados e como estes objetos obterão acessos entre si; e (3) implementação, que codifica o projeto em uma linguagem de programação.

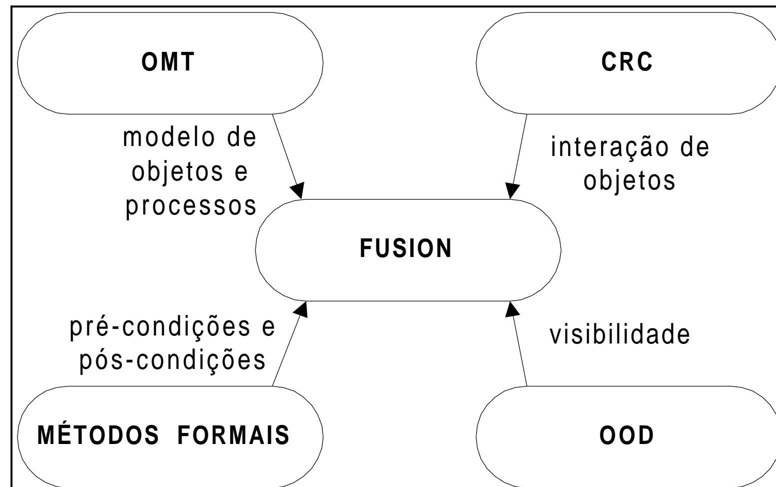

Figura 24 - Os componentes do método FUSION (COL96)

Como ilustrado na figura 25, o método não se preocupa com a obtenção dos requisitos do sistema, estes requisitos serão informados pelo usuário em um documento informal escrito, possivelmente, em linguagem natural.

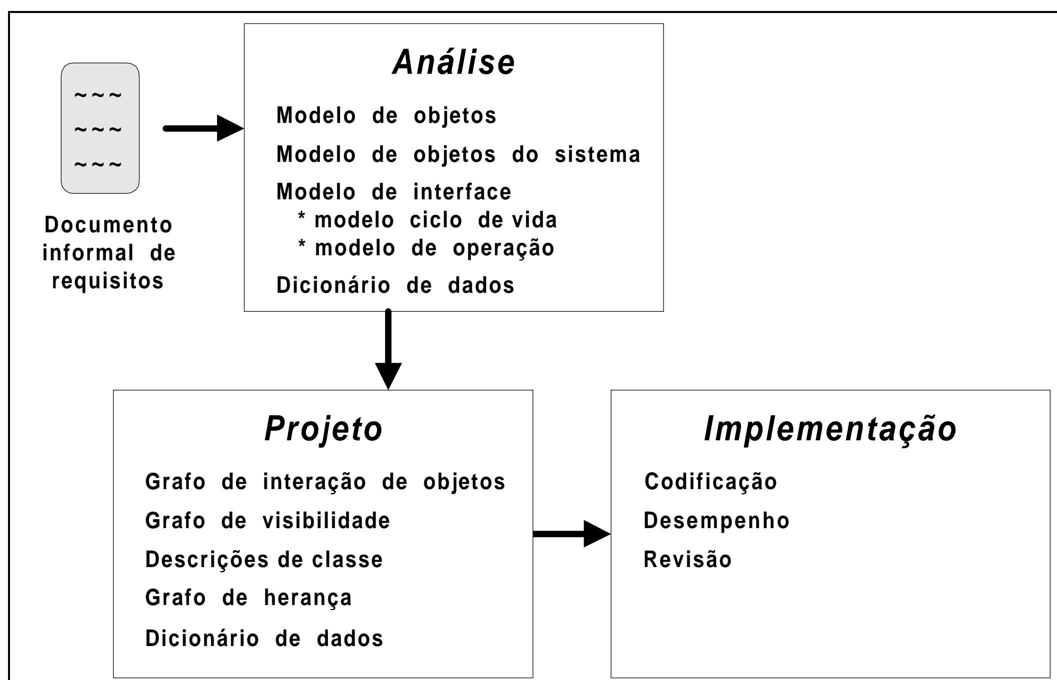

Figura 25 - As fases do método FUSION

Durante a execução das fases do FUSION, será necessário construir e utilizar um dicionário de dados, que será o repositório central das definições de termos e conceitos. O dicionário de dados representa uma ferramenta vital, devido a seu papel de único local onde as definições podem ser encontradas, auxiliando o entendimento de pessoas que não estejam familiarizadas com o desenvolvimento em questão.

São descritas a seguir, as fases do FUSION: análise, projeto e implementação.

1) Análise

O objetivo da análise é capturar a maior quantidade possível dos requisitos do sistema, de forma completa, consistente e sem ambigüidades. As informações que serão consideradas para análise encontram-se em um documento de definição de requisitos, escrito em linguagem natural.

Esta fase gera cinco modelos que capturam aspectos diferentes de um sistema:

(1) modelo de objetos, que define os conceitos existentes no domínio do problema e os relacionamentos existentes entre eles, podendo representar classes, atributos e relacionamento entre classes e as extensões que permitem o uso de agregação e generalização;

(2) modelo de objetos do sistema, que representa um subconjunto de um modelo de objetos relacionado ao sistema a ser construído. Será formado com a exclusão de todas as classes e relacionamentos pertencentes ao ambiente do sistema. Assim, um modelo de objetos do sistema possui um conjunto de classes e relacionamentos, possivelmente redundante, para a modelagem dos estados relativos ao sistema. Este modelo é um refinamento do modelo de objetos desenvolvido na primeira etapa da análise. Ele utiliza as informações relativas à interface do sistema para indicar quais classes e relacionamentos pertencem ao estado do sistema, em oposição àquelas que pertencem ao ambiente. Um modelo de objetos do sistema é um modelo que mostra a fronteira do sistema.

(3) modelo de interfaces, que descreve o comportamento (entrada e saída) do sistema, sendo que a descrição será feita em termos de eventos e mudanças de estado por eles causados. Um modelo de interfaces utiliza dois modelos[28] para capturar aspectos diferentes do comportamento de um sistema:

(3.1) modelo de operações, que especifica o comportamento das operações do sistema de forma declarativa, definindo seus efeitos em termos de mudanças de estado e eventos gerados. Ele define a semântica de cada operação do sistema que faz parte da

[28] A ordem do desenvolvimento não é fixa. Entretanto, será mais apropriado iniciar pelo ciclo de vida, pois ele pode auxiliar no desenvolvimento dos esquemas para o modelo de operações [COL96].

interface do sistema. Cada esquema contém uma especificação informal das pré-condições e pós-condições.

(3.2) modelo de ciclo de vida, que descreve o comportamento visto de uma perspectiva mais ampla, mostrando como o sistema se comunica com seu ambiente desde o momento da sua criação até o seu término. Ele determina se uma operação do sistema terá que enviar eventos aos agentes.

2) Projeto

Durante o projeto são incorporadas estruturas de *software* que satisfazem às definições abstratas produzidas durante a análise. O resultado do projeto é uma coleção de objetos que interagem entre si de modo a realizar o modelo de operações.

Há um total de quatro modelos de projeto que são desenvolvidos durante esta fase:

(1) grafos de interação de objetos, que constróem as estruturas que viabilizam as trocas de mensagens entre os objetos, realizando a definição abstrata do comportamento. Construi-se um grafo de interação de objetos para cada operação do sistema.

(2) grafos de visibilidade, que decide como os caminhos de comunicação serão realizados no sistema. Seu objetivo é o de definir as estruturas de referência entre as classes existentes no sistema. Há uma série de decisões de projeto que precisam ser tomadas quando consideramos a estrutura de referências para as classes. As decisões podem ser organizadas nas seguintes categorias: tempo de vida da referência, visibilidade do servidor, vinculação do servidor e mutabilidade das referências.

(3) descrições de classes, são coletadas informações existentes no modelo de objetos do sistema, nos grafos de interação de objetos e nos grafos de visibilidade para criar as descrições de classes, havendo uma descrição para cada classe. Aqui são estabelecidos para cada classe: as operações, atributos de dados e atributos-objetos. As descrições de classes fornecem os fundamentos para a implementação.

(4) grafos de herança, o propósito é desenvolver as estruturas de herança para o sistema. Os grafos de herança são construídos para todas as classes, com novas classes sendo introduzidas ao organizarmos a estrutura de classes em camadas abstratas. Finalmente, caso se tenha em mãos uma estrutura estável para o grafo de herança, o projetista pode atualizar as descrições de classes para que reflitam as novas classes abstratas e as estruturas de herança.

3) Implementação

Esta fase envolve o mapeamento do projeto em uma implementação efetiva. A base utilizada na fase de implementação é composta pelas descrições de classes, grafos de

interação de objetos e o dicionário de dados, todos produzidos na fase de projeto, incluindo também o ciclo de vida gerado durante a fase de análise. O processo de implementação pode ser dividido em três partes, cada uma delas com própria subestrutura:

(1) codificação, esta parte da fase de implementação, traduz o resultado obtido no projeto em código na linguagem de implementação, isto envolve quatro elementos: o ciclo de vida, as descrições de classes, os corpos dos métodos e o dicionário de dados;

(2) desempenho, utilizado quando um sistema consome recursos em demasia, deve-se proceder ao levantamento do seu perfil de desempenho, para descobrir onde o recurso é consumido. Uma vez localizados os 'gargalos', sejam eles de espaço, tempo ou qualquer outro recurso, o código pode ser melhorado.

(3) revisão, uma vez produzido o código, será preciso revisá-lo. O objetivo das inspeções[29] e dos testes[30] é detectar erros, antes que sejam passados para o código de produção. O código deve ser inspecionado durante sua produção. Isso permite ao grupo de inspeção impor padrões desde o início da fase de implementação, assegurando que problemas de codificação não fiquem ocultos até que seja tarde demais.

7.6.1 Considerações finais sobre o método FUSION

O método Fusion defende que não se deve definir comportamentos (dinâmica) para os objetos na fase de análise. Objetos de análise também não possuem interfaces.

Fusion não apresenta nenhum mecanismo para representação dos estados possíveis dos objetos, o que nos outros métodos é possível através das máquinas de estados. Em alguns tipos de aplicações este tipo de representação faz falta, pois os estados de objetos individuais podem ser bastante significativos.

Este método não é indicado para equipes iniciantes em orientação a objetos, pois é um método de ampla cobertura, se tornando um método de aprendizado demorado. Também não é um método indicado para projetos de baixa complexidade, que poderão fazer uso de métodos mais simples.

É o método ideal para desenvolvedores que já utilizaram na prática métodos mais simples, e desejam evoluir para um método mais completo.

[29] As inspeções requerem que o código possa ser lido por outras pessoas que não sejam seu próprio autor.

[30] Os testes verificam o comportamento real do sistema, ou apenas de parte dele, em relação às previsões feitas com base nos requisitos e nas especificações.

7.7 Comparação entre os conceitos

Nesta seção, é feita a comparação entre os conceitos dos métodos apresentados, pois cada um destes métodos utilizam diferentes nomes para o mesmo conceito. Através da tabela 4, pode-se igualar os conceitos e obter uma melhor compreensão dos métodos.

Esta comparação será baseada nos nomes referentes aos conceitos que foram apresentados na seção 6 deste trabalho.

O conceito de objeto, não foi apresentado na tabela 4, devido ao fato de que todos os métodos nomeiam este conceito igualmente.

Tabela 4 - Comparação entre os conceitos utilizados pelos métodos

Conceitos	Métodos				
	OMT	**OOAD**	**OOSE**	**OOA-OOD**	**FUSION**
Classe	Classe	Classe	Classe / Objeto	Classe	Classe
Classe abstrata	Classe abstrata	Classe abstrata	Classe abstrata	Classe	Classe abstrata
Classe concreta	Classe concreta	Classe concreta	Classe concreta	Classe&objetos	-
Operação	Operação	Operação	Operação	Serviço	Operação
Agregação	Agregação	Agregação	Associação consiste de / agregação	Estrutura Todo-parte	Agregação
Atributo	Atributo	Campo	Atributo	Atributo	Atributo
Herança	Herança	Herança	Herança	Estrutura Gen-Espec - Hierarquia	Gen-Espec
Herança múltipla	Herança múltipla	Herança múltipla	Herança múltipla	Estrutura Gen-Espec - entrelaçamento	Generalização múltipla
Mensagem	Evento	Mensagem	Estímulo	Mensagem	Mensagem
Subsistema	Módulo	Categ. Classe	Subsistema	Assunto	-

Fonte: Baseado em [JAC92] - pp. 501

A partir da análise da tabela 4, podemos observar que quase todos os autores utilizaram o mesmo nome para o mesmo conceito, porém alguns os diferenciaram, possuindo uma pequena divergência. Exemplos são: 1) o que é evento para OMT (mensagens entre objetos), para FUSION significa a comunicação do usuário com o sistema (o que acontece externamente e ao que o sistema tem que reagir); 2) um conceito não ilustrado na tabela 4, e muito utilizado pelo método OOA-OOD, e somente por ele, é o conceito classe&objetos, significando a classe e os objetos desta classe.

Podemos concluir ainda, que o método FUSION não menciona determinados conceitos: classe concreta e subsistema.

Passamos agora a comparação entre os métodos.

7.8 Comparação entre os métodos

Assim como foi feito a comparação entre os conceitos, utilizados pelos métodos para uma melhor compreensão destes, será realizado uma comparação entre os vários aspectos capturados pelos métodos, também com a mesma finalidade, facilitar a compreensão. Esta comparação é ilustrada na tabela 5 e tabela 6.

Através da tabela 5 e da tabela 6, pode-se notar que em geral, os métodos capturam aspectos semelhantes, como: estrutura estática de objetos e classes, seus relacionamentos, troca de mensagens e a fase de implementação; mas que também possuem individualidades, aspectos que outros métodos não capturam.

Somente o método OOSE, se preocupa com a fase de teste; os métodos OMT e OOA-OOD, possuem suporte gráfico, importante para um desenvolvimento com maior rapidez.

Podemos notar, também, através da tabela 5 e da tabela 6, que aspectos importantes para o desenvolvimento, como: identificação de requisitos, subdivisão do sistema, suporte gráfico, estrutura de agregação e generalização-especialização e teste do sistema, não são suportados por todos os métodos.

Alguns métodos realizam a captura de aspectos não importantes para outros métodos: visão lógica, aspectos funcionais, interface do sistema, apresentar multi-tarefa, conexão de ocorrência e teste do sistema.

Legenda para a tabela:

- D. E. O. - Diagrama de Estado do Objeto.
- M. A. - Modelo de Análise.
- M. P. - Modelo de Projeto.
- O. O. - Orientação a Objetos.

Tabela 5 - Comparação entre os aspectos capturados pelos métodos

Aspectos capturados	MÉTODOS				
	OMT	**OOAD**	**OOSE**	**OOA-OOD**	**FUSION**
Identificação dos requisitos	Fase de análise	Atividade de conceitualização	Modelo de requisitos	-	Fase de análise
Interação do usuário c/ o sistema	Diagrama de fluxo de dados Ord.: diagrama de eventos	*Use case*	*Use case*	M. A. camada serviço M. P. componente interação humana	Modelo de objetos do sistema
Interface do sistema	Formato de interface	-	Descrições de interface	-	-
Estrutura estática de classes	Diagrama de classes	Diagrama de classes	Modelo de análise	M. A. camada classe&objetos	Modelo de objetos
Estrutura estática de objetos	Diagrama de objetos	Diagrama de objetos	Modelo de análise	M. A. camada classe&objetos	Modelo de objetos
Relacionamentos entre classes	Diagrama de classes	Diagrama de classes	Modelo de análise	M. A. camada classe&objetos	Modelo de objetos
Relacionamentos entre objetos	Diagrama de instâncias	Diagrama de objetos	Modelo de análise	M. A. camada classe&objetos	Modelo de objetos
Definição de atributos	Diagrama de objetos	Diagrama de classes	-	M. A. camada atributos	-
Defin. atributos de dados	-	-	-	-	Descrições das classes
Definição das operações	Diagrama de objetos	Diagrama de classes	-	M. A. camada serviços Diagrama de serviço	Descrições das classes
Troca de mensagens	Diagrama de fluxo de evento Ord.: diagrama de eventos	Diagrama de objetos Ord.: diagrama de interação.	Diagrama de interação e grafos de transição de estado	M.A. envio: camada classe&objeto	Grafos de interação de objetos
Estrutura de agregação	Diagrama de objetos	Diagrama de classes	Modelo de análise	M. A. camada estrutura	-
Estrutura de generalização	Diagrama de classes	Diagrama de classes	-	M. A. camada estrutura	Grafos de herança
Documentação de estrutura de herança	-	-	-	-	Descrições das classes

Tabela 6 - Comparação entre os aspectos capturados pelos métodos (Cont.)

Aspectos capturados	MÉTODOS				
	OMT	**OOAD**	**OOSE**	**OOA-OOD**	**FUSION**
Descrição de classes	Dicionário de estados	-	-	-	Descrições das classes
Apresentar concorrência	Diagrama de estado	Modelo de objeto	Diagrama de interação (estímulo inter-processo)	M. P. componente Gerenciamento de tarefas	-
Conexão de ocorrência	-	-	-	M. A. camada atributos	-
Sequência de mensagens	Diagrama de eventos	-	Diagrama de interação	-	Modelo ciclo-de-vida
Sequência de estados	Diagrama de estados	Diagrama de transição de estado	-	M. A. camada serviços D. E. O.	Modelo de operações
Serviços dependentes de estados	-	-	-	Tabela de serviço / estado	-
Referenciação entre objetos	-	-	M. A. associação de conhecimento	-	Grafos de visibilidade
Aspectos funcionais	Diagrama de fluxo de dados	-	Modelo *use case*	-	-
Visão física	Projeto do sistema	Diagrama de módulos e diagrama de processos	-	-	-
Visão lógica	-	-	Modelo de objeto do domínio do problema	-	-
Subdivisão do sistema	Diagrama de fluxo de dados	Diagrama de módulos	Subsistema (modelo de análise)	M. A. camada de assunto	-
Apresentar multi-tarefa	-	-	-	M. P. componente gerenciamento de tarefas	-
Implementação	Tradução do projeto em determinada linguagem	Atividade de evolução	Modelo de implementação	Recomenda-se o uso de linguagem O. O.	Codificação, desempenho, revisão
Teste do sistema	-	-	Teste de unidade, de integração e de sistema	-	-
Suporte gráfico	OMTool	-	-	OOATool	-

7.9 Cobertura dos métodos

Os cinco métodos orientados apresentados nesta seção, subitens 7.2, 7.3, 7.4, 7.5 e 7.6, cobrem diferentes fases do desenvolvimento. Por esta razão, foi criada a figura 26 onde são apresentadas as fases que cada método cobre.

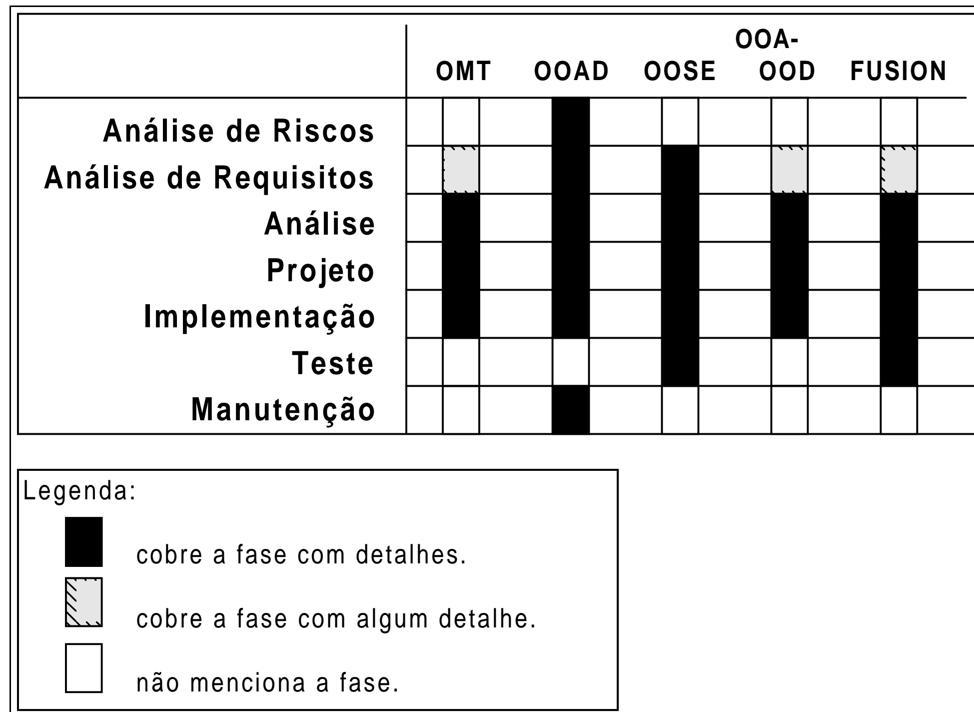

Figura 26 - Cobertura dos métodos

Através da análise da figura 26, pode-se perceber que a fase de análise de riscos é coberta somente pelo método OOAD, onde os riscos são analisados durante todo o desenvolvimento.

Somente os métodos OOAD e OOSE se preocupam com a fase de análise de requisitos e os métodos Fusion e OOA-OOD mencionam a fase, dizendo que obtém os requisitos através de um documento informal escrito pelo usuário, geralmente em linguagem natural.

Todos os métodos apresentados cobrem a fase de análise, projeto e implementação e, somente os métodos OOSE e Fusion descrevem a fase de teste[31].

O método OOAD cobre com detalhes a fase de manutenção, onde o método OMT cobre com algum detalhe.

[31] Acredita-se que durante a fase de implementação sejam executados alguns tipos de testes, mas que não são mencionados pelos autores dos métodos.

7.10 Considerações finais

Nesta seção foram apresentados os métodos orientados a objetos - OMT, OOAD, OOSE, OOA-OOD e FUSION -, descrevendo o processo de cada um e algumas considerações finais.

Foi realizada uma comparação entre os métodos, entre seus conceitos e entre os aspectos capturados por eles e a cobertura que cada um realiza. A partir do exposto pode-se concluir que cada método aborda diferentes aspectos, reforçando a idéia de que cada método será mais eficiente no desenvolvimento de determinados tipos de sistemas.

Para a finalização deste trabalho, a seguir são apresentadas as conclusões e as referências bibliográficas.

8. Conclusões

Foram apresentados conceitos da área de engenharia de *software*, uma descrição dos modelos de processo clássicos - *waterfall*, prototipação, espiral e transformacional -, dos conceitos relacionados a orientação a objetos e de cinco métodos orientados a objetos - OMT, OOAD, OOSE, OOA-OOD e FUSION.

Notamos que é necessário uma mudança de paradigma, pois a visão que a área de orientação a objetos tem para o desenvolvimento é bastante diferente da que muitos gerentes e desenvolvedores de *software* estão acostumados, que é o paradigma estruturado. Conceitos como: encapsulamento, objeto, classe e herança são novos e requerem um prazo para estudo e adaptação.

Uma comparação preliminar entre os métodos foi apresentada, em forma de uma tabela (tabela 5 e tabela 6), onde podemos concluir que todos os métodos abordam as principais características da orientação a objetos, mas se diferenciam com peculiaridades próprias, tais como: certas técnicas, diagramas ou modelos que os outros métodos não possuem.

Os métodos orientados a objetos foram recentemente criados e novos estão surgindo, o que aumenta o leque de opções para pesquisas e comparações. Os métodos relacionados na tabela 1, não abordam todas as fases do ciclo de desenvolvimento, e isto também poderia ser um estudo no sentido de melhorá-los.

Através do estudo feito, pode-se notar que os métodos possuem características que são comuns a todos, mas que também possuem características próprias que o classifica como o mais indicado para ser usado em determinados tipos de sistemas a serem desenvolvidos.

Para que a escolha do método e do modelo de processo seja adequada, é necessário o estudo das características do sistema, por exemplo: 1) se o sistema tem muita funcionalidade; 2) se os requisitos do sistema estão fracamente definidos; etc. Com bases em características deste tipo, poderemos escolher o método e o modelo de processo mais indicados ao desenvolvimento do sistema, e também deve ser observado que cada método é adequado para ser usado em conjunto com determinados modelos de processo. Mas, ainda se observa a necessidade de fazer um estudo das características da organização, por exemplo: a) se a equipe de desenvolvimento possui treinamento no método e modelo de processo escolhidos; b) se o prazo é adequado; etc.

Verifica-se então, a necessidade de um estudo mais aprofundado na área, onde iremos abranger os aspectos acima mencionados, que resultará em uma dissertação de tese. Este trabalho servirá como *background*, pois aqui foi feito um estudo geral da área.

Referências Bibliográficas

[**AND95**] **ANDERSSON**, M., **BERGSTRAND**, J.; Uses Cases in Object-Oriented Analysis. - Dissertação de Mestrado, submetida ao *Department of Communication Systems at Lund University, Sweden*, 1995. Disponível em: http://www.efd.lth.se/~d87man/ EXJOBB/Title_Abstract_Preface.html, 1995.

[**BAK96**] **BAKKER**, G.; OMT Object Model: Notation, Concepts and Constructs. - Disponível: http://www.comp.mq.edu.au/courses/comp331/OMT/notation.html, 1996.

[**BOO91**] **BOOCH**, G.; Object Oriented Design with Applications. - California: *The Benjamin/Cummings Publishing Company, Inc.*, 1991.

[**BOO94**] **BOOCH**, G.; Object Oriented Analysis and Design with Applications. - 2ª edição. - Canadá: Addison Wesley, 1994.

[**BOO96**] **BOOCH**, G.; Object Solutions: Managing the Object Oriented Project. - Menlo Park: Addison-Wesley, 1996.

[**BRI95**] **BRINKKEMPER**, S., **HONG**, S., **BULTHUIS**, A., **GOOR**, G. v.d.; Object Oriented Analysis and Design Methods – a Comparative Review. – *University of Twente*, 1995. Disponível em: http://wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo.html.

[**CAM93**] **CAMARÃO**, P. C. B.; Glossário de Informática. - Rio de Janeiro: LTC - Livros Técnicos e Científicos, 1993.

[**CAM97**] **CAMPOS**, F. B.; Sistematizando o Processo de Desenvolvimento de Software - Uma abordagem Orientada ao Reuso; Dissertação de mestrado submetida ao Departamento de Ciência da Computação da Universidade de Brasília. - Brasília, 1997.

[**CAP93**] **CAPRETZ**, L. F. e **LEE**, P. A.; *Object Oriented Design: guidelines and techniques*. - *Information and Software Technology*, Vol. 35, nº 4, ISSN 0950-5849, Abril, 1993.

[**COA92**] **COAD**, P., **YOURDON**, E.; *Análise Baseada em Objetos*. - Trad. da 2ª ed., CT Informática. - Rio de Janeiro: Editora Campus, 1992.

[**COA93**] **COAD**, P., **YOURDON**, E.; *Projeto Baseado em Objetos*; Trad. PubliCare Serviços de Informática, Vandenberg Dantas de Souza. - Rio de Janeiro: Editora Campus, 1993.

[**COL96**] **COLEMAN**, D., et al.; *Desenvolvimento Orientado a objetos: o método Fusion*; Trad. de Geraldo Costa Filho. - Rio de Janeiro: Editora Campus, 1996.

[**HOD95**] **HODGSON**, J., *Software Engineering Process Models*. - Disponível em: http://www.sju.edu/~jhodson/se/models.html, 1995.

[**HUM89**] **HUMPHREY**, W. S., *Managing Software Process*. - Canadá: Addison-Wesley, 1989.

[**JAC92**] **JACOBSON**, I., *Object-Oriented Software Engineering: A use case driven approach*. - USA: ACM Press, 1992.

[**KAN95**] **KAN**, S. H., *Metrics and Models in Software Quality Engineering*. - Massachusetts: Addison-Wesley, 1995.

[**KEY93**] **KEYES**, J., *Software Engineering Productivity Handbook*. - USA: McGraw Hill, 1993.

[**MAR**] **MARQUES**, A. F. D. M. R.; *Descrição de Metodologias de Análise orientadas a Objetos*. - Universidade do Minho, Braga, Portugal. Ano não identificado.

[**MAR94**] **MARTIN**, J.; *Princípio da Análise e Projeto baseado em Objetos*; Trad. de Cristina Bazán. - Rio de Janeiro: Editora Campus, 1994.

[**MON94**] **MONTGOMERY**, S. L.; *Information Engineering: analysis, design and implementation*. - Cambridge: Academic Press, 1994.

[**NAS90**] **NASCIMENTO**, A. R. C., *An Expert System to Advise on Information System Development Approaches*, Dissertação submetida a Universidade de Manchester, Departamento de Computação. - Inglaterra, 1990.

[**NAS92**] **NASCIMENTO**, M. E. M., *SMM (Software Management Model): A Multidimensional and Integrative Software Development Management Model*, Tese submetida a Universidade de Manchester, Departamento de Computação. - Inglaterra, 1992.

[**PER92**] **PEREIRA**, C. E., Métodos de Análise de Sistemas de Tempo Real usando Técnicas de Orientação a Objetos. - São Carlos: X Simpósio Brasileiro de Engenharia de *Software*, 14-18/outubro, 1996.

[**PRE92**] **PRESSMAN**, R. S., Engenharia de *Software*; Trad. de José Carlos Barbosa dos Santos. - São Paulo: Makron Books, 3a. edição, 1992.

[**REI94**] **REINOSO**, G. B. e **HEUSER** C. A., Comparando o Processo de Modelagem de Técnicas de Análise Orientada a Objetos. - Curitiba - PR: Anais do VIII Simpósio Brasileiro de Engenharia de *Software*, 25-28/outubro, 1994.

[**RUM94**] **RUMBAUGH**, J., et al.; *Modelagem e Projetos baseados em Objetos*; Trad. de Dalton Conde de Alencar. - Rio de Janeiro: Editora Campus, 1994.

[**SHL90**] **SHLAER**, S., **MELLOR**, S.J.; *Análise de Sistemas Orientada para Objetos*; Trad. de Anna Terzi Giova. - São Paulo: Editora McGraw-Hill, 1990.

[**SIL96**] **SILVA**, R. P. e ; *Avaliação de metodologias de análise e projeto orientadas a objetos voltadas ao desenvolvimento de aplicações, sob a ótica de sua utilização no desenvolvimento de frameworks orientados a objetos*, Monografia submetida a UFRGS. - Porto Alegre, 1996.

[**SOM92**] **SOMMERVILLE**, I., *Software Engineering*. - USA: Addison-Wesley, 4a. edição, 1992.

[**THA88**] **THAYER**, R. H., *Tutorial: Software Engineering Project Management*. - Los Alamitos: IEEE Computer Society Press, 1988.