

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Curso de Mestrado

**Problemas, Desafios e
Abordagens do Processo
de Desenvolvimento
de *Software***

Trabalho Individual I

Rafael Prikladnicki

Orientador
Prof. Dr. Jorge Luis Nicolas Audy

Porto Alegre, 13 de fevereiro de 2004.

SUMÁRIO

LISTA DE FIGURAS	iv
LISTA DE ABREVIATURAS E SIGLAS.....	v
1 INTRODUÇÃO	6
2 A Engenharia de <i>Software</i> e o Processo de Desenvolvimento de <i>Software</i>	11
3 Ciclos de Vida do Processo de Desenvolvimento de <i>Software</i>	16
3.1 Ciclo de Vida Clássico – Modelo Cascata	17
3.1.1 Principais Virtudes do Ciclo de Vida Clássico	18
3.1.2 Possíveis Problemas do Ciclo de Vida Clássico	18
3.2 Ciclo de Vida Incremental - Prototipação	18
3.2.1 Principais Virtudes do Ciclo de Vida Incremental.....	20
3.2.2 Possíveis Problemas do Ciclo de Vida Incremental	20
3.3 O Ciclo de Vida Espiral	20
3.3.1 Principais Virtudes do Ciclo de Vida Espiral	21
3.3.2 Possíveis Problemas do Ciclo de Vida Espiral	22
3.4 Combinando Modelos de Ciclo de Vida.....	22
3.4.1 Técnicas de Quarta Geração	23
4 Principais problemas do desenvolvimento de <i>software</i>	24
4.1 Dificuldades Acidentais e Dificuldades Essenciais	24
4.2 Problema Conceitual e a Especificação de Requisitos.....	25
4.3 Definição e cumprimento dos prazos.....	26
4.4 Custo de um projeto	27
4.5 Produtividade.....	27
4.6 Qualidade e Teste de <i>Software</i>	27
4.7 Trabalho em equipe.....	28
4.8 Capacitação de Pessoal	30
4.9 Planejamento.....	31
4.10Motivação.....	32
4.11Manutenibilidade	32
4.12Gerência de Projeto	34
4.13Análise Crítica	35

5	Principais desafios do desenvolvimento de <i>software</i>	38
5.1	Novos Ambientes de Desenvolvimento	38
5.1.1	E-Business e Desenvolvimento Web	38
5.1.2	Outsourcing	40
5.1.3	Ambientes fisicamente distribuídos	41
5.2	Gerenciamento de riscos	42
5.3	Capacitação de Pessoal	43
5.4	Planejamento	43
5.5	Padrões de Desenvolvimento de <i>Software</i>	44
5.6	Certificação	44
5.7	Aprendizagem Organizacional	45
5.8	Produtividade e Motivação	46
5.9	Conclusões	47
6	Abordagens de Desenvolvimento de <i>Software</i>	49
6.1	Evolução das Abordagens de Desenvolvimento de <i>Software</i>	49
6.2	Abordagem Estruturada	50
6.3	Modelagem de Dados	53
6.4	Orientação a Objetos	54
6.4.1	Conceitos Básicos de Orientação a Objeto	55
6.4.2	Análise e Projeto Orientado a Objeto	57
6.4.3	A UML (Linguagem Unificada de Modelagem)	58
6.4.4	Vantagens da Abordagem Orientada a Objeto	60
6.4.5	Os Problemas Potenciais da Orientação a Objetos	61
6.5	Abordagem Estruturada x Orientação a Objetos	61
6.5.1	Vantagens da OO em relação à Abordagem Estruturada	63
	CONCLUSÕES	64
	REFERÊNCIAS BIBLIOGRÁFICAS	65

LISTA DE FIGURAS

Figura 1 - A evolução do <i>Software</i>	7
Figura 2 - As camadas da Engenharia de <i>Software</i>	12
Figura 3 - Modelo de Ciclo de Vida Clássico.	17
Figura 4 - Modelo de Ciclo de Vida Incremental.	19
Figura 5 - Modelo de Ciclo de Vida Espiral.	21
Figura 6 - Aplicação das técnicas de Quarta Geração.	23
Figura 7 - Distribuição dos esforços de manutenção.	33
Figura 8 - Categorias de problemas do desenvolvimento de <i>software</i>	36
Figura 9 - Gerência de risco segundo Boehm, 1989.	42
Figura 10 - Modelo de motivação em um conjunto de níveis.	47
Figura 11 - Evolução das abordagens de desenvolvimento de <i>software</i>	49
Figura 12 - Evolução da UML.	50
Figura 13 - Engenharia da Informação.	54
Figura 14 - Análise e Projeto Orientados a Objeto.	58
Figura 15 - Modelos conceituais em cada abordagem.	62

LISTA DE ABREVIATURAS E SIGLAS

4GT – *4 Generation Techniques.*

a.C. – antes de Cristo.

Case – *Computer-Aided Software Engineering.*

CMM – *Capability Maturity Model.*

EUA – Estados Unidos da América.

PMI – *Project Management Institute.*

OMG – *Object Management Group.*

OMT – *Object Modeling Technique.*

OOSE – *Object Oriented Software Engineering.*

OO – Orientação a Objetos.

OOPSLA - *Object Oriented Programming, Systems, Languages and Applications.*

SGBDs – Sistemas de Gerência de Banco de Dados.

SO – Sistema Operacional.

TI1 – Trabalho Individual 1.

UML – *Unified Modeling Language.*

1 INTRODUÇÃO

Quando se iniciava a década de 1980, uma reportagem de primeira página da revista *Business Week* trazia a seguinte manchete: “*Software: A Nova Força Propulsora*”. O *software* estava amadurecendo, tornando-se um tema de preocupação da área de negócios nas organizações. Em meados da década de 1980, uma reportagem de capa da *Fortune* lamentava “Uma Crescente Defasagem de *Software*” e, ao final da década, a *Business Week* avisava os gerentes sobre “A Armadilha do *Software* – Automatizar ou Não”. No começo da década de 1990, uma reportagem especial da *Newsweek* perguntava: “Podemos Confiar em Nosso *Software*?”, enquanto o *Wall Street Journal* relacionava as “dores de parto” de uma grande empresa de *software* com um artigo de primeira página intitulado “Criar *Software* Novo: Era Uma Tarefa Agonizante...”. Essas manchetes e muitas outras iguais a elas eram o anúncio de uma nova compreensão da importância do *software* de computador, as oportunidades que ele oferece e os perigos que ele representa [PRE 95].

Durante as três primeiras décadas da Era do computador, o principal desafio era desenvolver um *hardware* que reduzisse o custo de processamento e armazenagem de dados. Ao longo da década de 1980, avanços na microeletrônica resultaram em maior poder de computação a um custo cada vez mais baixo. Já na década de 1990, o principal desafio foi melhorar a qualidade (e reduzir o custo) de soluções baseadas em computador – soluções que são implementadas com *software*. [PRE 95].

Hoje, o problema é diferente. O mundo da engenharia de *software* vem se desenvolvendo cada vez mais rápido. É necessário desenvolver *software* de altíssima qualidade. Temos à nossa disposição diversas ferramentas novas e, muito poderosas, que nos ajudam a organizar o processo de desenvolvimento [PET 01].

Esta evolução do *software* dentro do contexto das áreas de aplicação de sistemas baseados em computador pode ser descrita desde a década de 1950 [PRE 95]. Durante os **primeiros anos** do desenvolvimento de sistemas computadorizados, o *hardware* sofreu contínuas mudanças, enquanto o *software* era visto por muitos como uma reflexão posterior. A programação de computador era uma arte "secundária" para a qual haviam poucos métodos sistemáticos. O desenvolvimento do *software* era feito virtualmente sem método, até que os prazos comesçassem a se esgotar. Durante esse período, era usada uma orientação *batch* (em lote) para a maioria dos sistemas. A **segunda Era** da evolução dos sistemas computadorizados estendeu-se da década de 1960 até o final da década de 1970. A multi-programação e os sistemas multi-usuários introduziram novos conceitos de interação homem-máquina. Esta segunda Era também foi caracterizada pelo uso do produto de *software* e pelo advento das "*software houses*". O *software* era desenvolvido para ampla distribuição num mercado interdisciplinar. A **terceira Era** da evolução dos sistemas computadorizados começou em meados da década de 1970, onde os sistemas distribuídos aumentaram intensamente a complexidade dos sistemas baseados em computador. Esta Era também foi caracterizada pelo advento uso de microprocessadores, computadores pessoais e poderosas estações de trabalho de mesa. A **quarta Era** do *software* de computador trouxe as tecnologias orientadas a objeto, que rapidamente ocuparam o lugar das abordagens mais convencionais para o desenvolvimento de *software* em muitas áreas de aplicação.

A figura 1 mostra a evolução do *software* ao longo dos anos.

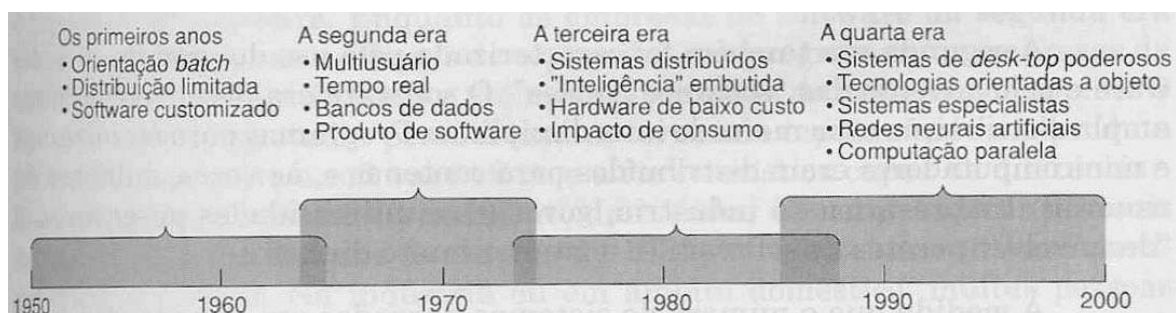


Figura 1 - A evolução do *Software*.
Fonte: [PRE 95].

Quando nos movimentamos para a **quinta Era**, os problemas associados ao desenvolvimento de *software* continuam a se intensificar:

- A sofisticação do *software* ultrapassou a capacidade de construir um *software* que extraia o potencial do *hardware*.
- A capacidade de construir programas não pode acompanhar o ritmo da demanda de novos programas.
- A capacidade de manter os programas existentes é ameaçada por projetos ruins e recursos inadequados.

É interessante observar como alguns dos princípios fundamentais desse desenvolvimento permanecem os mesmos de 30 anos atrás e como os desafios que os engenheiros de *software* encontram hoje são semelhantes aos existentes quando a engenharia de *software* ainda engatinhava. Após 30 anos, ainda temos que nos esforçar para desenvolver *software* confiáveis [PET 01].

Além dos problemas ainda presentes no processo de desenvolvimento de *software*, atualmente novos desafios são apresentados. Um dos mais significativos destes desafios envolve o processo de desenvolvimento de *software* em ambientes fisicamente distribuídos. Hoje em dia as grandes organizações estão cada vez mais distribuindo seus processos de desenvolvimento de *software* ao redor do mundo, visando ganhos de produtividade, redução de custos, diluição de riscos e melhorias na qualidade. Ou seja, o *software* está ficando cada vez mais sofisticado, o processo de desenvolvimento de *software* está evoluindo de uma forma muito rápida e, além disso, se vê a necessidade de distribuir o desenvolvimento, trazendo muitos benefícios para as organizações.

Neste contexto, países como Índia, Irlanda e Brasil emergem como potenciais candidatos a sediarem estes centros de desenvolvimento de *software* mundiais. No caso brasileiro, a legislação específica na área de informática incentiva as empresas, através da redução de impostos, a aplicarem recursos em projetos de pesquisa aplicada cooperados com Universidades e Centros de Pesquisa locais. Diversas organizações de grande porte na área de informática estão se beneficiando destes

incentivos e direcionando seus esforços para a criação de centros de desenvolvimento de *software* para atender demandas da organização em sua sede, tipicamente nos EUA e na Europa [PRI 02].

Neste cenário, destaca-se o surgimento de uma nova classe de problemas no processo de desenvolvimento de *software*, que envolve as diferenças culturais e a distância física entre os participantes do processo. Desta forma, os já tradicionais problemas inerentes ao processo de desenvolvimento, fortemente centrados nas fases de especificação de requisitos e análise de sistemas, ganham contornos mais críticos [PRI 02]. A forma de resolver estes problemas está centrada na adoção de linguagens de especificação e processos de desenvolvimento mais formais e definidos. Modelos de verificação e certificação do nível de maturidade do processo de desenvolvimento de *software*, tipo *CMM (Capability Maturity Model)*, têm-se tornado cada vez mais importante e úteis para as organizações contratantes terem um mínimo de garantia sobre a qualidade do processo utilizado pela organização parceira. A Era das abordagens de desenvolvimento monolíticas e pouco formais está no final. Os desenvolvedores estão cientes da existência de múltiplas formas de especificar e desenvolver sistemas [MAR 98]. Novas tecnologias e tipos de sistemas de informação, tais como *e-business*, sistemas especialistas, redes neurais, algoritmos genéticos e máquinas de inferências e regras demandam diferentes abordagens de desenvolvimento [PRI 02].

Em outras palavras, se existem diferentes formas através da qual um sistema pode ser especificado [LAR 00], mesmo considerando um mesmo paradigma de desenvolvimento (p.ex., orientação a objeto), porque não escolher a melhor combinação do uso destas abordagens para um determinado tipo de problema ou ambiente? Por que ficar preso a um modelo ou metodologia única e rígida, se uma combinação de abordagens, técnicas e ferramentas pode gerar um resultado mais preciso, adequado, econômico e elegante?

Neste contexto, o Trabalho Individual 1 (TI1) está direcionado para a área de engenharia de *software* no sentido de estudar alguns aspectos do processo de

desenvolvimento de *software*, identificando algumas de suas abordagens, problemas e desafios.

O capítulo 2 traz uma breve introdução sobre a Engenharia de *Software*, bem como uma breve contextualização do cenário atual do processo de desenvolvimento de *software*. O capítulo 3 apresenta os modelos de ciclos de vida de desenvolvimento de *software*. A seguir, o capítulo 4 aborda os principais problemas que existem no desenvolvimento de *software*. O capítulo 5 enfoca quais os principais desafios existentes, considerando a realidade atual. Por último, o capítulo 6 tem por objetivo identificar as principais abordagens de desenvolvimento de *software*, fazendo uma comparação entre as abordagens mais antigas e as atuais.

2 A Engenharia de Software e o Processo de Desenvolvimento de Software

O principal desafio na área de Engenharia de *Software* nas últimas duas décadas tem sido o estudo e a melhoria da qualidade e redução de custo do *software* produzido [PRE 01]. De um modo geral, pode-se entender a Engenharia de *Software* como sendo um conjunto de disciplinas. A literatura apresenta diversas definições, e algumas delas são apresentadas a seguir:

"A Engenharia de Software é o estabelecimento e o uso de bons princípios de engenharia e boas práticas de gerenciamento, além da evolução de ferramentas e métodos para uso apropriado, a fim de obter, dentro dos limites existentes, um software de alta qualidade [MAC 90]".

"Engenharia de Software pode ser definida pelo estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais [PRE 01]".

"Engenharia de Software é uma disciplina que reúne metodologias, métodos e ferramentas a serem utilizados, desde a percepção do problema até o momento em que o sistema desenvolvido deixa de ser operacional, visando resolver problemas inerentes ao processo de desenvolvimento e ao produto de software [CAR 01]".

Ainda que muitas definições abrangentes tenham sido propostas, todas elas convergem no sentido de apontar para necessidades de maior rigor no desenvolvimento de *software*. Sendo assim, a partir da revisão bibliográfica e baseando-se nas definições anteriormente citadas, encontrou-se na definição de [IEE 93] uma forma mais abrangente para definir a Engenharia de *Software*. Esta definição diz que:

“Engenharia de Software é a aplicação de um ambiente sistemático, disciplinado e quantificável para o desenvolvimento, operacionalização e manutenção do software; ou seja, a aplicação da engenharia ao software. [IEE 93]”.

Segundo [PRE 01], a engenharia de *software* pode ser entendida através de camadas. Estas camadas abrangem três elementos fundamentais: ferramentas, métodos e processo. De acordo com a figura 2, cada um destes elementos corresponde a uma camada, sendo que a camada base representa o foco na qualidade. Isto significa que os elementos representados nas três primeiras camadas devem ser capazes de possibilitar ao gerente o controle do processo de desenvolvimento de *software* e oferecer ao desenvolvedor uma base para a construção de *software* de alta qualidade.



Figura 2 - As camadas da Engenharia de *Software*.
Fonte: [PRE 01].

Os 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, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste, manutenção, etc.

As ferramentas de engenharia de *software* proporcionam apoio automatizado ou semi-automatizado aos métodos. Atualmente, existem ferramentas para sustentar cada um dos métodos citados anteriormente. Quando as ferramentas são integradas de forma que a informação criada por uma ferramenta possa ser usada em outra, é

estabelecido um sistema de suporte ao desenvolvimento de *software* chamado Engenharia de Software Auxiliada por Computador (*CASE – Computer-Aided Software Engineering*).

Segundo [PRE 01], o processo é a camada mais importante da engenharia de *software*. Esta camada consitue o elo de ligação que mantém juntos as ferramentas e os métodos, além de possibilitar um desenvolvimento racional do *software*. Um processo define a seqüência em que os métodos serão aplicados, como os produtos serão entregues, os controles que ajudam a assegurar a qualidade e a coordenar as mudanças, e os marcos de referência que possibilitam aos gerentes de *software* avaliar o progresso do desenvolvimento [PRE 01].

Um processo de desenvolvimento de *software* é representado por um modelo, enquanto que o modelo é operacionalizado por meio de uma metodologia. Existem diversos modelos de processo de desenvolvimento de *software*, e cada modelo pode ter mais de uma metodologia que o operacionaliza. A metodologia estabelece basicamente a seqüência das atividades e como elas se relacionam entre si, identificando o momento em que os métodos e as ferramentas serão utilizados.

Sendo assim, um processo de desenvolvimento de *software* deve ser implementado de acordo com um modelo previamente definido, seguindo uma metodologia que se adeque às necessidades e objetivos existentes, e tudo isto deve servir de guia para a correta utilização dos métodos e das ferramentas, tendo sempre em mente que a camada básica é o foco na qualidade.

Com relação à qualidade, existem alguns princípios da engenharia de *software* que descrevem de maneira geral as propriedades desejáveis para um produto de *software* [CAR 01]. Entre estes princípios, pode-se citar:

- **Formalidade:** por ser uma atividade criativa, o desenvolvimento de *software* tende a ser não estruturado, pois depende da “inspiração do momento”. Mas através de uma sistemática formal, é possível produzir produtos mais confiáveis, controlar o seu custo e ter mais confiança no seu desempenho. A formalidade não deve restringir a criatividade, mas deve melhorá-la;

- **Abstração:** abstração é o processo de identificação dos aspectos importantes de um determinado fenômeno, ignorando-se os detalhes. Podem existir diferentes abstrações da mesma realidade, cada uma fornecendo uma visão diferente da realidade e servindo para diferentes objetivos;

- **Decomposição:** uma das maneiras de trabalhar com a complexidade é subdividir o processo em atividades específicas, atribuídas à especialistas de diferentes áreas. Esta separação permite o planejamento das atividades e diminui o tempo extra que seria gasto mudando de uma atividade para outra. Além do processo, o produto também pode ser desenvolvido através de sub-produtos, definidos de acordo com o sistema que está sendo desenvolvido. Entre as vantagens desta decomposição está a execução das atividades de forma paralela. E o objetivo maior é diminuir a complexidade;

- **Generalização:** a generalização pode ser boa, mas ao mesmo tempo pode trazer algumas desvantagens no desenvolvimento de um produto de *software*. Uma das vantagens é a possibilidade da reutilização de uma solução em diversos pontos do sistema. Mas uma solução genérica é bem mais custosa em termos de velocidade de execução ou tempo de desenvolvimento. Sendo assim, o importante é saber avaliar se vale a pena desenvolver uma solução generalizada, dependendo do sistema que está sendo desenvolvida e dos custos envolvidos;

- **Flexibilidade:** quando fala-se em flexibilidade fala-se na possibilidade de um produto ser modificado com facilidade. O processo deve ter flexibilidade suficiente para permitir que componentes do produto desenvolvido possam ser utilizados em outros sistemas, e deve-se avaliar também a sua portabilidade para diferentes sistemas computacionais.

Todos estes princípios sozinhos não são suficientes para guiar o desenvolvimento de *software*. Eles devem ser aplicados dentro de um contexto onde exista um ambiente de desenvolvimento de *software* bem definido, utilizando um modelo e uma metodologia adequadas, com métodos e ferramentas de apoio.

Cabe salientar que não existe um modelo de processo ideal. A escolha depende do tamanho da organização, da experiência existente dentro da equipe, da natureza e da complexidade da aplicação, do prazo de entrega, entre outros fatores. Existem diversos modelos de processos disponíveis e prontos para serem utilizados. Entretanto, cada vez mais busca-se adequar um modelo a um cenário específico. Além disso, deve existir um modelo de ciclo de vida fortemente conectado ao modelo de processo de desenvolvimento de *software* escolhido.

3 Ciclos de Vida do Processo de Desenvolvimento de *Software*

Como foi descrito no capítulo anterior, a engenharia de *software* compreende um conjunto de camadas que envolvem métodos, ferramentas e processo. Qualquer desenvolvimento de um produto inicia com uma idéia e termina com o produto pretendido. O ciclo de vida de um *software* é a definição dos passos que transformam aquela idéia no produto acabado [PET 01]. Sendo assim, este capítulo tem como objetivo apresentar os principais modelos de ciclos de vida de desenvolvimento de *software*.

Os modelos de ciclo de vida são o centro do processo de gerenciamento do *software*. Estes modelos possibilitam ao gerente controlar o processo de desenvolvimento de *software* e permite ao desenvolvedor obter a base para produzir de maneira eficiente um *software* que satisfaça os requisitos estabelecidos. Os ciclos de vida especificam algumas atividades que devem ser executadas, assim como a sua ordem. Sua função básica é diminuir os problemas encontrados no processo como um todo.

Devido à importância do ciclo de vida, vários modelos já foram propostos. A escolha de um modelo adequado é crítica. Um ciclo de vida deve ser escolhido tendo-se como base a natureza do projeto e da aplicação, os métodos e as ferramentas a serem usados, além dos controles e os produtos que precisam ser entregues. O ciclo de vida deve viabilizar uma definição dos pontos de controle, o planejamento e acompanhamento do progresso e do orçamento, uma estimativa de tempo de desenvolvimento, além de uma boa gerência de risco.

A seguir serão apresentados os principais modelos de ciclo de vida que são utilizados atualmente nos processos de desenvolvimento de *software*. Todos eles

funcionam, de acordo com a característica do projeto. O segredo está na escolha do modelo certo ou na união de alguns deles.

3.1 Ciclo de Vida Clássico – Modelo Cascata

Às vezes chamado de modelo cascata, é um ciclo de vida que utiliza um método sistemático e seqüencial, em que o resultado de uma fase se constitui na entrada de outra. A figura 3 mostra o modelo deste ciclo de vida. Cada fase é estruturada como um conjunto de atividades que podem ser executadas por pessoas diferentes, simultaneamente.

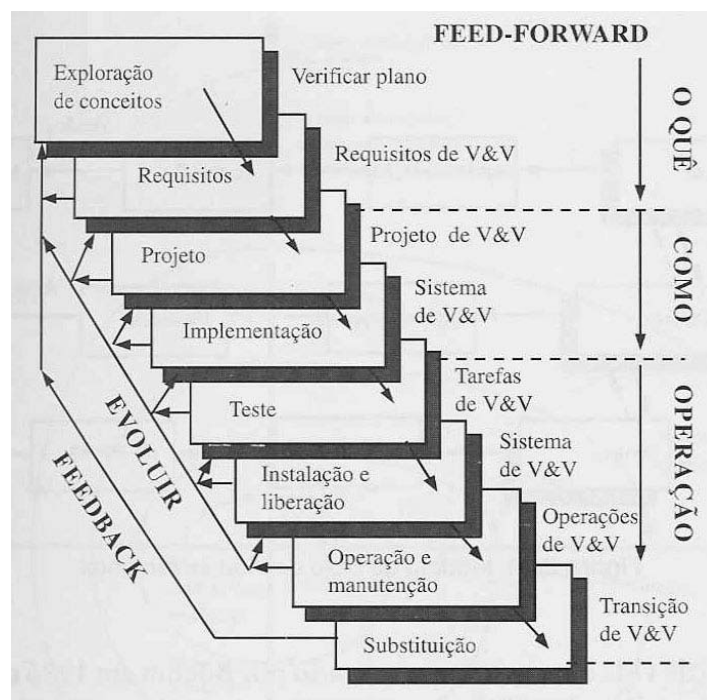


Figura 3 - Modelo de Ciclo de Vida Clássico.
Fonte: [PET 01].

Segundo [PRE 95] e [PET 01], o ciclo de vida clássico abrange atividades de:

- **Análise e engenharia de sistema:** quanto mais dados forem coletados em nível de sistema, menor será a probabilidade de haver "*bugs*" e, conseqüentemente, poderá diminuir os futuros reparos;
- **Análise de requisitos de software:** é importante saber o que o cliente quer que o *software* tenha, com relação aos recursos. Os requisitos devem ser documentados e revistos com o cliente antes de começar a execução do projeto;

- **Projeto:** envolve muitos passos que se concentram em 4 atributos distintos do programa: estrutura de dados, arquitetura de *software*, detalhes de procedimentos e caracterização de interface;

- **Codificação:** o projeto deve ser traduzido de forma legível para uma linguagem de máquina;

- **Testes:** deve-se testar todas as instruções a procura de erros. O resultado real deve concordar com o projeto ou resultado exigido;

- **Manutenção:** indubitavelmente o *software* sofrerá mudanças depois que foi entregue ao cliente e poderá sofrer *upgrades* de tempos em tempos.

3.1.1 Principais Virtudes do Ciclo de Vida Clássico

Segundo [CAR 01], o ciclo de vida clássico possui as seguintes virtudes:

- Caracteriza fases estanques para as quais podem ser descritas técnicas para o seu desenvolvimento de forma clara;
- É o ciclo de vida mais amplamente utilizado da engenharia de *software*;

3.1.2 Possíveis Problemas do Ciclo de Vida Clássico

Mesmo sendo o ciclo de vida mais antigo e o mais usado em engenharia de *software*, o ciclo de vida clássico pode apresentar alguns problemas, tais como:

- Os projetos reais raramente seguem o fluxo seqüencial que o modelo propõe;
- O cliente não saberá declarar todas as suas exigências;
- O cliente deve ter paciência, pois qualquer erro detectado após a revisão do programa de trabalho pode ser desastroso.

Os problemas são reais, mas mesmo assim, o ciclo de vida clássico continua sendo o mais utilizado no desenvolvimento de *software*.

3.2 Ciclo de Vida Incremental - Prototipação

Muitas vezes, as dúvidas do cliente e do programador em relação ao *software* levam ao processo de prototipação, também chamado de ciclo de vida incremental. A

prototipação capacita o desenvolvedor a criar um modelo de *software* que será implementado. A figura 4 mostra o modelo deste ciclo de vida.

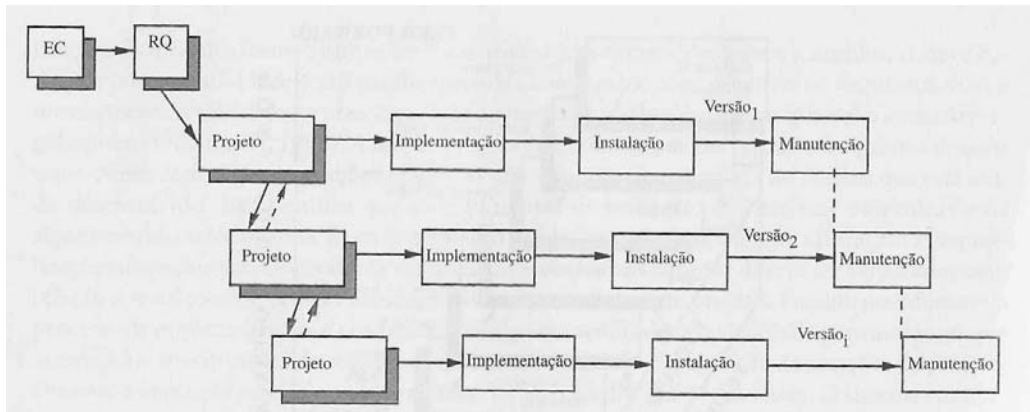


Figura 4 - Modelo de Ciclo de Vida Incremental.
Fonte: [PET 01].

De acordo com [PRE 95] e [PET 01], a prototipação pode assumir uma das três formas abaixo citadas:

- Um protótipo em papel ou no computador, que retrata a interação homem máquina, podendo ver quanta interação haverá;
- Um protótipo que implemente funções específicas, da função exigida pelo *software*;
- Um programa existente que executa parte ou toda a função desejada, mas que tem características que deverão ser melhoradas.

O ciclo de vida incremental abrange as atividades de:

- **Coleta de requisitos:** da mesma forma que ocorre em todos os ciclos de desenvolvimento de *software*;
- **Elaboração de um projeto:** deve ser rápido, contendo os aspectos que serão visíveis ao cliente. O projeto rápido leva à construção de um protótipo, que será avaliado pelo cliente e pelo usuário;
- **Avaliação do protótipo:** esta avaliação será usada para refinar requisitos para o *software* desenvolvido.

Idealmente, o protótipo serve como um mecanismo para identificar os requisitos de *software*. Muitas vezes é preciso descartar um protótipo e partir do início para evitar perda de tempo com correções.

3.2.1 Principais Virtudes do Ciclo de Vida Incremental

Segundo [PRE 01], o ciclo de vida incremental possui as seguintes virtudes:

- Permite que o usuário interaja de forma mais ativa na modelagem do sistema;
- Facilita a identificação dos requisitos do *software*;
- Acelera o desenvolvimento do sistema.

3.2.2 Possíveis Problemas do Ciclo de Vida Incremental

O ciclo de vida incremental pode apresentar problemas pelas seguintes razões:

- O cliente quer resultados e muitas vezes não saberá, ou não entenderá, que um protótipo pode estar longe do software ideal, que ele nem sequer imagina como é. Mesmo assim, a gerência de desenvolvimento cede às reclamações e tenta encurtar o prazo de entrega, o qual já estava prolongado;
- O desenvolvedor, na pressa de colocar um protótipo em funcionamento, é levado a usar um *SO* ou linguagem de programação imprópria por simplesmente estar a disposição ou estar mais familiarizado. Essa atitude poderá levar a soluções ineficientes.

Ainda que possam ocorrer problemas, o uso da prototipação é bastante eficiente na engenharia de *software*. O segredo é o entendimento entre desenvolvedor e cliente.

3.3 O Ciclo de Vida Espiral

Também conhecido como paradigma de Bohem [BOE 91], foi desenvolvido para englobar as melhores características dos ciclos de vida clássico e incremental, ao mesmo tempo em que adiciona um novo elemento, a **análise de risco**, que não existe nos modelos anteriores. Sendo assim, este ciclo de vida define quatro importantes atividades, de acordo com [PRE 95], [PET 01]:

- **Planejamento**: determinação de objetivos, alternativas e restrições;

- **Análise dos riscos:** análise de alternativas e identificação / resolução dos riscos;

- **Engenharia:** desenvolvimento do produto;

- **Avaliação do cliente:** avaliação dos resultados da engenharia.

Baseado principalmente em decisões de prosseguir / não prosseguir, de acordo com a avaliação, seja do cliente ou do desenvolvedor, o modelo espiral tende a uma trajetória que rumo para o modelo mais completo do sistema.

O ciclo de vida de modelo espiral é atualmente a abordagem mais realística para o desenvolvimento de *softwares* e sistemas em grande escala. Ele usa uma abordagem "evolucionária", capacitando o desenvolvedor e o cliente a entender e reagir aos riscos, em cada etapa evolutiva da espiral. A figura 5 mostra o modelo deste ciclo de vida.

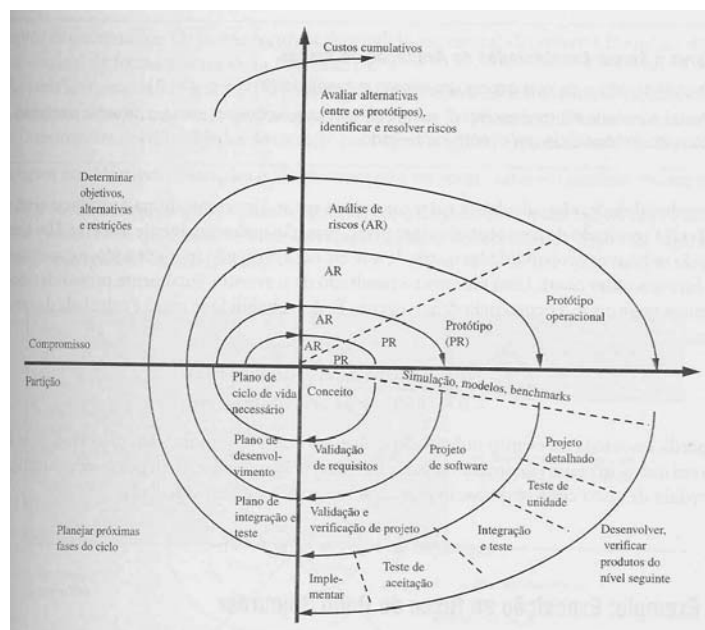


Figura 5 - Modelo de Ciclo de Vida Espiral.
Fonte: [PET 01].

3.3.1 Principais Virtudes do Ciclo de Vida Espiral

O ciclo de vida espiral possui as seguintes virtudes [PRE 95], [PET 01], [SEI 02]:

- Permite um desenvolvimento evolutivo do *software*;
- Permite uma interação com o usuário;
- Os requisitos não precisam ser todos definidos no começo;

- É Iterativo, com um marco para avaliação ao final de cada iteração;
- Busca integração do desenvolvimento tradicional com a prototipação;
- Introduz a análise de risco.

3.3.2 *Possíveis Problemas do Ciclo de Vida Espiral*

O ciclo de vida espiral tem como principal característica a busca de um mecanismo de redução de riscos, e este mecanismo é aplicado em qualquer ponto evolutivo do ciclo. Porém, podem surgir problemas, tais como:

- Pode ser difícil convencer grandes clientes de que a abordagem evolutiva é controlável;
- Se um grande risco não for descoberto, com certeza ocorrerão problemas;
- É mais complexo e tem um custo mais alto do que os outros ciclos de vida;
- Pode não convergir para uma solução;
- Dependendo do projeto, a relação custo / benefício pode ser duvidosa.

Esse modelo é relativamente novo, e não tem sido amplamente usado como os dois ciclos anteriormente explicados. Somente o tempo, que gera a experiência, poderá comprovar a eficácia do modelo espiral.

3.4 **Combinando Modelos de Ciclo de Vida**

Em muitos casos, no processo de desenvolvimento de *software*, os modelos de ciclo de vida podem e devem ser combinados de forma que as potencialidades de cada um possam ser obtidas num único projeto. O modelo espiral já faz isso diretamente, combinando o modelo incremental com elementos do ciclo de vida clássico em um ciclo de vida evolutivo. Entretanto, qualquer modelo pode constituir a base sob a qual os outros poderão ser integrados [CAR 01].

O processo sempre começa com a determinação dos objetivos, alternativas e restrições, que algumas vezes é chamada de obtenção de requisitos preliminares. Depois disso, qualquer caminho pode ser tomado. Por exemplo, os passos do ciclo de vida clássico podem ser seguidos se o sistema puder ser completamente especificado no começo. Se os requisitos não estiverem muito claros, um protótipo pode ser usado

para melhor defini-los. Usando o protótipo como guia, o desenvolvedor pode retornar aos passos do ciclo de vida clássico (projeto, implementação e teste). De forma alternativa, o protótipo pode evoluir para um sistema, retornando em cascata para ser testado. Além disso, alguns marcos podem ser instanciados para verificar como o desenvolvimento do *software* está em um determinado momento, e uma análise de riscos pode ser feita visando evitar surpresas ao final do desenvolvimento. A natureza da aplicação é que vai determinar o modelo a ser utilizado, e a combinação de modelos só tende a beneficiar o processo como um todo.

3.4.1 Técnicas de Quarta Geração

Um fator interessante é descrito por [PRE 01], quando ele cita as técnicas de quarta geração (4GT) como um outro modelo de ciclo de vida possível de ser aplicado em um processo de desenvolvimento de *software*. Atualmente o processo de desenvolvimento de *software* evoluiu consideravelmente, permitindo que as técnicas de quarta geração possam ser utilizadas em conjunto com qualquer outro modelo de ciclo de vida citados anteriormente. O que se percebe é que as técnicas de 4GT fazem parte de uma dimensão horizontal no contexto dos ciclos de vida, podendo ser utilizadas em qualquer um destes modelos, e não na dimensão vertical, compondo mais um modelo diferente dos modelos já citados (como está representado na figura 6).

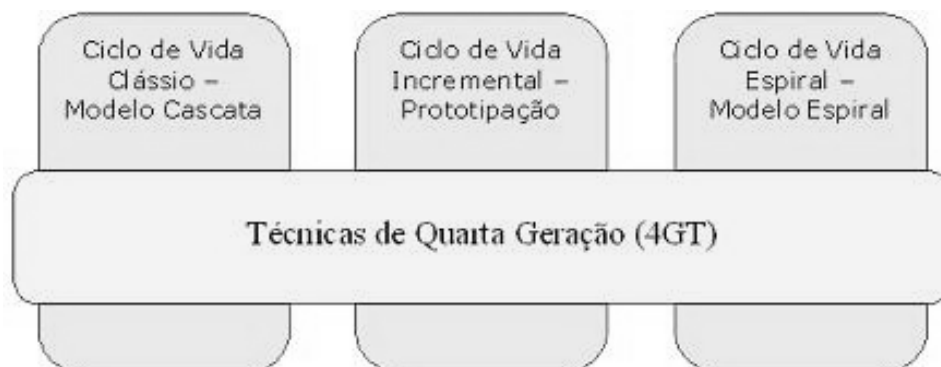


Figura 6 - Aplicação das técnicas de Quarta Geração.

4 Principais problemas do desenvolvimento de *software*

Os problemas que afligem o desenvolvimento de *software* podem ser caracterizados a partir de uma série de perspectivas diferentes. Desta forma, neste capítulo o objetivo é abordar os principais problemas que são citados na literatura, abrangendo os problemas em todos os níveis do processo de desenvolvimento, incluindo desde recursos humanos até decisões referentes à área de negócios.

A partir da ampla revisão bibliográfica feita ([AUD 01], [CAR 01], [MAC 90], [PET 01], [PRE 95], [PRE 01], [PRI 02], [ROY 98], [SOM 95], [TEI 79], entre outros) identificou-se diversos problemas inerentes ao processo de desenvolvimento de *software*. A seguir serão destacados os principais problemas identificados.

4.1 Dificuldades Acidentais e Dificuldades Essenciais

Quando fala-se em dificuldades acidentais e dificuldades essenciais, é necessário exemplificar utilizando-se recursos de algum objeto. Sendo assim, a distinção entre os recursos acidentais e essenciais de um objeto foi apresentada por Aristóteles em um livro chamado Tópicos, escrito por volta de 340 a.C. [PET 01].

Um recurso é considerado acidental se a existência de algo não depender do recurso. Exemplos de recursos acidentais de *software* são a sua linguagem (alto nível ou nível de máquina), sua representação gráfica ou composição (modular ou não), etc. Esses recursos podem ser modificados sem que a essência do *software* seja alterada. Como exemplo, a Orientação a Objeto pode ser considerada um recurso acidental de um projeto de *software*. A funcionalidade de um programa orientado a objeto é mantida mesmo se ele for reescrito de forma não-orientada a objeto. Da mesma forma a construção conceitual implícita em um programa permanece a mesma se este contém um único bloco ou é modularizado. A funcionalidade de um programa

também é mantida se ele ficar mais compreensível ao ser escrito em uma linguagem de alto nível como C++, ao invés de ser escrito em linguagem de máquina, como *assembly*.

Por outro lado, um recurso é considerado essencial se algo não puder ser mantido sem ele. Complexidade e abstração são exemplos de dificuldades essenciais das entidades de *software*. A complexidade pode ser medida em relação ao número de predicados condicionais de um programa. Ela também pode ser medida com base em tipos de instruções, contagem de operadores, níveis de alinhamento, fluxo de informações e contagem de instruções [PET 01].

Em suma, um problema básico que ocorre no desenvolvimento de *software* é decidir o que desejamos dizer, e não como dizê-los. Com isso, as dificuldades acidentais podem ser superadas. Além disso, várias abordagens já foram sugeridas para especificar, projetar e testar a construção implícita em um *software* em desenvolvimento. O refinamento iterativo e interativo dos requisitos com o auxílio de uma prototipação rápida é visto como uma forma de atacar a essência conceitual do software [PET 01]. Um protótipo de *software* simula interfaces selecionadas e realiza uma ou mais funções principais do sistema. A vantagem da prototipação de *software* é que ela tende a revelar uma estrutura conceitual específica, de forma que o cliente possa testar sua consistência e usabilidade. Outra forma de atacar as dificuldades essenciais do *software* é desenvolvê-lo de forma incremental [PET 01].

4.2 Problema Conceitual e a Especificação de Requisitos

Segundo [PET 01], foram identificados dois problemas principais no desenvolvimento de *software*: um problema conceitual e um problema representativo. O problema conceitual envolve todo o desenvolvimento relacionado à especificar, projetar e testar a construção conceitual implícita em um sistema de *software*. Este é um problema considerado difícil, pois a essência de uma entidade de *software* é uma construção de conceitos inter-relacionados. Esses conceitos podem ser encontrados

nos conjuntos de dados, nas relações entre os itens de dados, nos algoritmos e nas chamadas de funções dentro de um programa.

O problema representativo envolve a representação do *software*, e o teste da fidelidade de uma representação. É um problema considerado mais fácil, pois está ligado a recursos acidentais do *software*.

Além destes problemas, existe um grande obstáculo no desenvolvimento de *software* que é a coleta e a especificação de requisitos. Na maioria das empresas, onde não existe um processo formal definido, não existe um tempo suficiente para fazer uma coleta de requisitos profunda sobre um determinado sistema, o que faz com que o *software* seja desenvolvido em um tempo muito maior do que o planejado. Além disso, muitas vezes não existe uma preocupação em formalizar determinados procedimentos, bem como uma discussão sobre os requisitos para se chegar em uma solução. Tudo é feito ao mesmo tempo, o que também impede a coleta de métricas e prejudica a produtividade. Na prática, algumas vezes o problema ocorre pois faltaram alguns requisitos, e em outras vezes os requisitos existem mas foram especificados de forma insuficiente à contemplar todas as funcionalidades [PRE 01].

4.3 Definição e cumprimento dos prazos

Freqüentemente as estimativas de prazos dos projetos de *software* são definidas de forma bastante imprecisas, e isto reflete diretamente na sua entrega, podendo ocorrer atrasos e problemas com o cliente [McC 96]. Isto ocorre devido principalmente a inexistência de um tempo adequado para coletar dados sobre o processo de desenvolvimento de *software*. Com poucos dados, as estimativas de prazos acabam não sendo reais, tendo como resultado uma estimativa bastante ruim. Além disso, muitas vezes os projetos são mal dimensionados, prevendo muito pouco tempo ou tempo em demasia [PRE 01]. E quando não existe uma indicação sólida de produtividade, não existe uma possibilidade precisa de avaliar a eficácia de novas ferramentas, métodos ou padrões.

4.4 Custo de um projeto

O custo de um projeto envolve a avaliação e definição de estimativas que contemplam o esforço que será gasto para o desenvolvimento de um determinado *software*. O custo pode ser expresso através de algumas variáveis (dias, horas, etc.), e sobre estas variáveis se estipula um valor que será cobrado pelo desenvolvimento. [PMB 00]. Mas da mesma forma que ocorre na definição de prazos, muitas vezes não existem dados suficientes para se determinar um custo adequado. Sendo assim, muitas vezes os projetos acabam sendo mal avaliados e seu custo pode ser um obstáculo na hora de fechar um contrato com um cliente [PET 01].

4.5 Produtividade

A produtividade é uma peça chave no processo de desenvolvimento de *software*. Por definição, significa rendimento, facilidade de produzir. E muitas vezes a produtividade das pessoas da área de *software* não têm acompanhado a demanda por seus serviços [PRE 01]. Como consequência, a insatisfação do cliente com o sistema concluído ocorre muito freqüentemente. Os projetos normalmente são executados de qualquer maneira, com um vago indício das necessidades do cliente. A comunicação entre o cliente e o desenvolvedor muitas vezes é insuficiente. Falta uma sistemática onde seja possível interagir para captar todas as informações necessárias e para a produtividade se tornar um diferencial de uma área de desenvolvimento de *software* [McC 96].

4.6 Qualidade e Teste de Software

Hoje em dia muitas pessoas falam em qualidade de *software*, mas nem sempre as pessoas têm uma noção clara desse conceito [COR 01]. E é isto que faz a qualidade e o teste de *software* serem considerados problemas do processo de desenvolvimento de *software*. Muitas vezes a qualidade dos sistemas é insuficiente. Isto ocorre pois a prática de testes de *software* é recente, e apenas agora sua importância está sendo realmente entendida. O mesmo ocorre com a verificação da qualidade como um todo,

do início ao fim do processo de desenvolvimento de *software* [SCH 99]. Apenas agora estão começando a surgir conceitos quantitativos de confiabilidade e garantia de qualidade de *software*. Além disso, um *software* construído sem qualidade pode ser muito difícil para manter. Na maioria das vezes a manutenção é deixada de lado pois o cliente não pensa neste aspecto quando quer desenvolver um sistema. Cabe ao desenvolvedor a tarefa de saber avaliar o tempo de vida e o futuro do *software* que será construído, fazendo um produto realmente de qualidade [PRE 01], que esteja de acordo com os requisitos do sistema e com o processo de desenvolvimento previamente definido.

Quando uma empresa diz que é a favor da qualidade sem tomar nenhuma atitude nesse sentido, nada acontecerá. Dizer que a qualidade é a prioridade número um não exerce um impacto real sobre o trabalho que está sendo desenvolvido, a menos que isso esteja combinado com alguma ação real de melhoria de qualidade. A maioria das empresas têm como alvo a entrega do *software* no prazo. E infelizmente isso, em geral, significa que o cliente pode receber um produto defeituoso no prazo. Existem muitas dificuldades com relação aos mecanismos que garantam a qualidade do *software*. Certificação é uma opção, criar uma equipe formal de garantia de qualidade é outra opção e existem diversas outras que podem ser citadas. Mas a cultura da empresa deve ser alterada para aplicar com sucesso mecanismos de prevenção de defeitos e de melhoria do processo de desenvolvimento [COR 01].

4.7 Trabalho em equipe

Muitos engenheiros de *software* são motivados principalmente pelo seu trabalho. Equipes de desenvolvimento de *software* normalmente são compostas por pessoas que têm suas próprias idéias em como solucionar problemas técnicos específicos. Mas muitas vezes isto é prejudicial, causando problemas que poderiam ser melhor conduzidos e solucionados se conduzidos em um ambiente de trabalho em grupo.

Por isso, um grupo com personalidades e conhecimentos complementares pode trabalhar melhor como equipe do que um grupo que foi formado apenas levando em

consideração as habilidades técnicas de seus integrantes [SCH 00]. As pessoas que são motivadas pelo seu trabalho geralmente têm um perfil técnico muito forte. Por outro lado, outros perfis profissionais são relevantes, tais como profissionais focados em resultado (que levam o trabalho ao seu final), sabendo que existem restrições de prazos, custos e escopo que nem sempre permitem que a melhor solução técnica seja implantada. É muito importante que haja uma interação positiva entre os profissionais de uma equipe, potencializando os conhecimentos complementares existentes, levando à melhores resultados e evitando conflitos [SOM 95].

Mas as vezes é impossível montar uma equipe com personalidades e conhecimentos complementares. Neste caso, aumenta a importância de uma efetiva gerência do processo de trabalho nas áreas técnicas. Busca-se, através deste processo de gerência do pessoal, garantir que os objetivos do grupo se sobreponham aos objetivos individuais.

[SOM 95] sugere que os grupos devem ser coesos e pequenos, e cita algumas vantagens, tais como:

- Pode ser desenvolvido um padrão de qualidade do grupo, pois este padrão é estabelecido por todos;
- Os membros da equipe trabalham juntos e podem aprender uns com os outros;
- Os membros da equipe podem saber o que cada um está desenvolvendo no seu trabalho e podem trocar informações;
- O desenvolvimento de determinada atividade passa a ser do grupo como um todo, e não é propriedade de algum membro do grupo apenas.

Além disso, o mesmo [SOM 95] também cita alguns problemas de um grupo ser pequeno e coeso:

- Resistência com relação à mudança de um líder de um grupo. Neste caso, quando um grupo que está bem integrado tem o seu líder trocado, a tendência é que ocorra uma união entre os membros deste grupo contra o novo líder. Pode haver uma resistência para a mudança, e perda de

produtividade. Por isso, sempre que possível, novos líderes devem surgir de dentro do próprio grupo;

- O grupo pode criar uma identidade muito forte, inibindo a ocorrência de discussões.

A maneira como a estrutura está organizada, como estão divididas as equipes, quem faz parte de cada equipe, são pontos que devem ser levados em consideração no desenvolvimento de *software*, pois muitos problemas podem surgir da diferença entre as pessoas, da comunicação entre as equipes, e principalmente, quando os interesses pessoais se sobrepõem aos interesses do grupo.

4.8 Capacitação de Pessoal

A natureza lógica do *software* constitui um desafio para as pessoas que o desenvolvem. O desafio intelectual do desenvolvimento de *software* é uma das causas de aflição que o afeta, mas a maioria dos problemas são causados por falhas humanas do dia-a-dia [PRE 01].

Para ilustrar, pode-se citar alguns exemplos do que acontece na prática, onde gerentes de nível médio e superior sem nenhum conhecimento em *software* recebem a responsabilidade pelo seu desenvolvimento. Segundo [PRE 95], existe um antigo axioma da administração que afirma que *“um bom gerente pode gerenciar qualquer projeto”*. Mas isto pode ser totalmente válido *“...se ele estiver disposto a aprender quais são os marcos (milestones) que podem ser usados para medir o processo, aplicar métodos efetivos de controle, não levar em conta o mito e ter um grande conhecimento em uma área que se modifica rapidamente”*. O gerente deve se comunicar com todas as pessoas envolvidas no desenvolvimento (clientes, desenvolvedores, pessoal de apoio, entre outros), buscando atuar como agente conciliador entre os atores do processo de desenvolvimento de *software*.

Além disso, os profissionais da área de *software* (programadores no passado e agora engenheiros de *software* ou desenvolvedores) têm recebido pouco treinamento formal em novas técnicas para desenvolvimento [McC 96]. Cada pessoa aborda a

tarefa de “escrever programas” com a experiência vivida em outras organizações. Algumas pessoas desenvolvem através de tentativa e erro, enquanto outras desenvolvem praticas inadequadas que se refletem diretamente na qualidade e manutenibilidade do *software*.

Todo mundo resiste a mudanças, mas se todos parassem para pensar que enquanto o potencial de computação (*hardware*) experimenta novas e enormes mudanças em um curto espaço de tempo, as pessoas da área de *software* responsáveis pelo aproveitamento deste potencial muitas vezes se opõem à mudança quando ela é discutida e resistem à mudança quando ela é introduzida. Esta pode ser uma das causas dos problemas enfrentados pelo desenvolvimento de *software* [SOM 95].

4.9 Planejamento

O ponto central de qualquer esforço para o desenvolvimento de um *software* é especificar, projetar e testar a construção conceitual do *software* que está sendo criado. O planejamento é o ponto de partida de qualquer atividade relacionada ao desenvolvimento de *software*. Muitas vezes o planejamento é esquecido, o que aumenta consideravelmente as chances de ocorrer algum problema [AUD 01]. As principais dificuldades estão relacionadas diretamente com a gestão do projeto e sua organização.

A ausência de uma etapa formal de planejamento pode ser apontada como um dos principais problemas no processo de desenvolvimento de *software*, diluindo decisões críticas ao processo como um todo em etapas subseqüentes onde perde-se a dimensão sistêmica da aplicação ou do problema em análise [AUD 01], [REP 98].

James Martin, em 1991, já apontava para a necessidade de inserção desta etapa no processo de desenvolvimento de *software* [MAR 91]. O modelo proposto por este autor considerava a existência de quatro grandes etapas no processo de desenvolvimento de *software*: planejamento, análise do negócio, projeto do *software* e construção. Seus estudos apontavam que a ausência de maior rigor nesta etapa de

planejamento acarretava um grande número de problemas nas etapas subseqüentes. Abordagens mais recentes [AUD 01], [BOA 97], [ROY 98] incorporam a etapa de planejamento no processo de desenvolvimento de software, visando se antecipar a possíveis problemas que a ausência desta etapa podem causar. Entretanto, ênfases de base mais tecnicistas tendem a abandonar esta perspectiva, se concentrando em aspectos mais intrínsecos às etapas mais técnicas do desenvolvimento de *software* em si. Mais recentemente, [PRE 01] retoma este problema ao afirmar a importância e a necessidade da etapa de planejamento no processo de desenvolvimento de *software*.

4.10 Motivação

Motivação, por definição, é o ato de dar motivo, ser causa de alguma coisa, despertar o interesse por alguma atividade [SCH 00]. Estar motivado significa que, além de ter um conhecimento e uma habilidade técnica, é necessário ter interesse em desempenhar determinada função. Uma pessoa motivada pode ter muita facilidade em produzir. Por outro lado, uma pessoa desmotivada pode ser o começo de muitos problemas no desenvolvimento de *software*. Em suma, a falta de motivação pode ocasionar diversos problemas dentro de um contexto de desenvolvimento de *software*, refletindo diretamente na produtividade, na qualidade e principalmente na responsabilidade [McC 96].

4.11 Manutenibilidade

A manutenção de *software* é um conceito antigo e extremamente importante, e que muitas vezes não tem sua importância reconhecida. [PRE 01] utiliza-se de algumas referências para caracterizar a manutenção de *software* como um "iceberg", ou seja, sabe-se que uma massa enorme de problemas e custos potenciais esconde-se sob a superfície.

Por definição, a manutenção é muito mais do que consertar erros. Existem três atividades básicas que devem ser realizadas após um determinado programa ser colocado em produção [PRE 01, SOM 95]:

- **manutenção corretiva**, ou seja, diagnóstico e correção de um ou mais erros;
- **manutenção adaptativa**, ou seja, uma atividade que modifica o *software* para que ele tenha uma interface adequada com o ambiente mutante;
- **manutenção perfectiva**, ou seja, à medida que o *software* é utilizado, recomendações de modificações em funções existentes e de ampliações gerais são recebidas dos usuários.

[PRE 01] ainda sugere uma quarta atividade chamada de **manutenção preventiva**, ou seja, ocorre quando o *software* é modificado para melhorar a confiabilidade ou a manutenibilidade futura, ou para oferecer uma base melhor para futuras ampliações. Essa atividade é caracterizada pelas técnicas de engenharia reversa e reengenharia.

A figura 7 mostra a distribuição dos esforços das atividades citadas anteriormente:

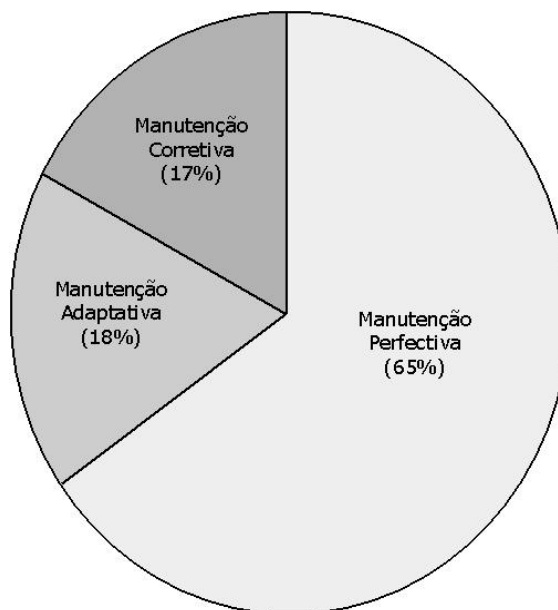


Figura 7 - Distribuição dos esforços de manutenção.
Fonte: [SOM 95].

Por muito tempo a manutenção era uma fase negligenciada do processo de engenharia de *software*. A falta de controle e disciplina nas atividades de desenvolvimento quase sempre se traduz em problemas durante a manutenção do *software*. Entre os diversos problemas, [PRE 01] destaca os principais, entre eles:

- É difícil e às vezes impossível rastrear a evolução do *software* através de muitas versões ou lançamentos. As mudanças não estão corretamente documentadas;
- É difícil e às vezes impossível rastrear o processo através do qual o *software* foi desenvolvido;
- A documentação não existe ou é muito ruim. Reconhecer que um *software* deve ser documentado é o primeiro passo, mas toda a documentação gerada deve estar de acordo com a lógica adotada;
- A maioria dos *softwares* não são projetados para sofrer mudanças;
- A manutenção não é vista como um trabalho muito importante. Grande parte desta percepção vem do elevado nível de frustração associado ao trabalho de manutenção;

Todos estes problemas podem ser atribuídos ao grande número de sistemas existentes que foram desenvolvidos sem levar em consideração alguma metodologia, regras, etc. De um modo geral, a engenharia de *software* atualmente oferece diversas soluções para cada problema associado à manutenção.

4.12 Gerência de Projeto

Por definição, a gerência de projetos é a aplicação de conhecimentos, habilidades e técnicas para projetar atividades que visem atingir ou exceder as necessidades e expectativas das partes envolvidas, com relação ao projeto [PMB 00].

O conhecimento sobre gerência de projetos e seus problemas podem ser organizados de muitas formas. [PMB 00] divide a gerência de projetos em algumas áreas de conhecimento, descrevendo os conhecimentos e práticas de gerência em termos dos processos que as compõe. [SCH 00] apresenta um modelo para gerência de projeto de sistemas de informação baseado no modelo do *Project Management Institute* (PMI), incorporando as nove áreas do conhecimento propostas em [PMB 00]. Estas áreas são: Gerência de Integração de Projeto, Gerência de Escopo do Projeto, Gerência de Qualidade do Projeto, Gerência dos Recursos Humanos, Gerência das

Comunicações do Projeto, Gerência dos Riscos do Projeto e Gerência das Aquisições do Projeto. Este modelo é proposto visando atuar sobre os problemas que a autora identifica em projetos na área de sistemas de informação nas organizações [SCH 00]:

- O usuário não se envolve no processo de desenvolvimento;
- Falta um suporte para a gerência de projetos;
- Os requisitos não são bem definidos;
- O planejamento não reflete a realidade;
- As expectativas não são realistas;
- Não são definidos marcos de avaliação nos projetos;
- A equipe não tem competência suficiente para desenvolver o projeto;
- Os objetivos não são claros;

A gerência de projeto tem uma importância fundamental para o sucesso do desenvolvimento de *software*. Todos os problemas citados até o momento podem ser tratados, ou ao menos identificados, se o projeto for bem gerenciado. Uma má gerência de projeto pode significar a perda do projeto e dos recursos que estão envolvidos [ROY 98]. Gerenciar projetos não é uma tarefa fácil, e é apenas com experiência que uma pessoa pode se tornar um especialista nesta área.

4.13 Análise Crítica

Além dos problemas citados anteriormente, identificaram-se na literatura alguns outros problemas, tais como aceitação do *software* pelo usuário, falta de comunicação, tempo de desenvolvimento muito longo (obsolescência de *hardware* e *software*), documentação inexistente, incompleta ou desatualizada. Devido ao grande número de problemas identificados, buscou-se apresentá-los de uma forma sintetizada, identificando um conjunto de categorias para agrupar estes problemas.

[GLA 98] propõe uma lista bastante interessante para as principais causas de falhas no processo de desenvolvimento de *software*. Esta classificação pode ser vista logo a seguir:

1. Objetivos do projeto não estão totalmente claros e especificados;

2. Má elaboração do planejamento e da estimativa;
3. A organização incorporou uma nova tecnologia;
4. Não existe um método de gerência de projeto, ou o método existente é inadequado;
5. Falta de uma equipe com experiência suficiente para desenvolver o projeto;
6. O equipamento existente na organização apresenta uma baixa performance de *hardware* e *software*;
7. Outros problemas de performance e/ou eficiência.

Buscando sintetizar a categorização proposta por [GLA 98] com os problemas identificados na revisão teórica desenvolvida (seções 3.1 a 3.12), consolidou-se um esquema de representação dos problemas, agrupando-os por temas convergentes.

A figura 8 apresenta a proposta de agrupamento de todos os problemas levantados, classificando-os em categorias mais amplas.

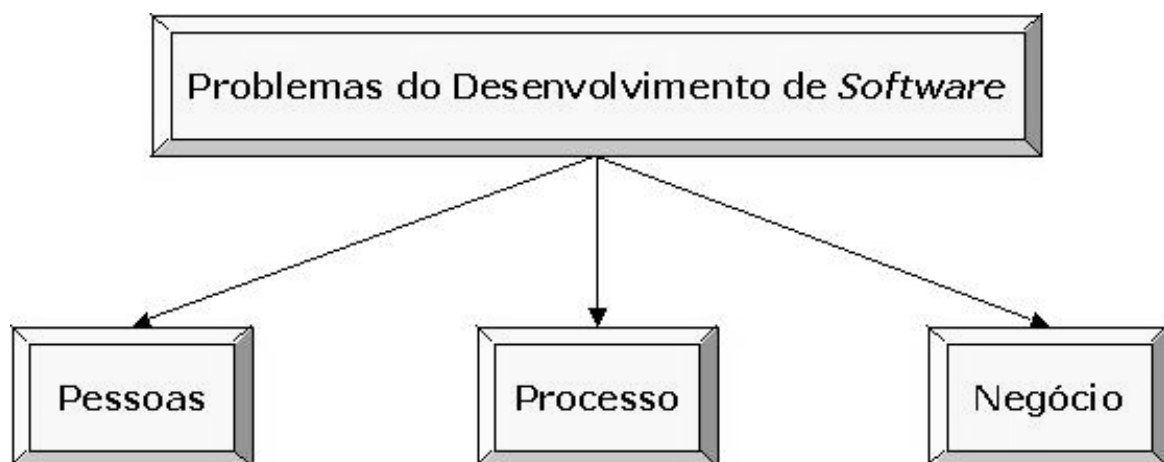


Figura 8 - Categorias de problemas do desenvolvimento de *software*.

De acordo com os principais problemas anteriormente apresentados, as categorias propostas agrupam os seguintes problemas:

Categoria	Problemas	Autores relacionados
Pessoas	Capacitação de Pessoal Motivação Produtividade Trabalho em Equipe	[McC 96], [PRE 95], [PRE 01], [SOM 95] [McC96], [SCH 00] [McC96] [SCH 00], [SOM 95]
Processo	Dificuldades Acidentais Dificuldades Essenciais	[PET 01] [PET 01]

	Especificação de Requisitos Qualidade do <i>Software</i> Manutenibilidade	[PET 01], [PRE 01] [COR 01], [PRE 01], [SCH 99] [PRE 01], [SOM 95]
Negócio	Custo Gerência de Projeto Planejamento Prazo	[PET 01], [PMB 00] [PMB 00], [ROY 98] [AUD 01], [PRE 01], [REP 98] [McC 96], [PRE 01]

5 Principais desafios do desenvolvimento de *software*

Os principais desafios do desenvolvimento de *software* hoje em dia estão fortemente ligados com a maioria dos problemas relatados na literatura, os quais foram tratados no capítulo anterior. Este capítulo aborda os principais desafios identificados na revisão teórica desenvolvida.

5.1 Novos Ambientes de Desenvolvimento

5.1.1 E-Business e Desenvolvimento Web

Velocidade, mudanças contínuas e competição global estão forçando as empresas a fazerem uma redefinição radical dos seus processos para sobreviverem. Fusões e aquisições estão cada vez mais freqüentes, novas tecnologias e mercados são diariamente criados. Os clientes estão ficando cada vez mais exigentes, sofisticados e seletivos. A consequência disso é o investimento em novas tecnologias e formas de fazer negócios, como o desenvolvimento de *Web sites*.

Na virada do século, já existiam mais de 400 mil empresas em todo o mundo investindo nesta nova forma de negócio através da Internet, para alcançar consumidores e parceiros a qualquer hora e em qualquer lugar. A denominação dada para este tipo de negócio é *e-business*, o qual engloba qualquer tipo de negociação feita através de meio eletrônico, seja entre empresa e cliente final (*business to consumer*) recebendo a denominação de *e-commerce* ou entre empresas (*business to business*) [STE 00].

Dada a relevância cada vez maior da internet na vida das pessoas e a tendência de uma sociedade cada vez mais voltada para o mundo digital, as empresas estão se reestruturando de maneira a adequar-se a esta nova realidade. E estes novos ambientes de desenvolvimento tem sido um dos grandes desafios das empresas.

O *e-business* propicia várias vantagens competitivas às empresas, dentre elas [ALB 00]:

- **Conveniência:** como a cadeia de valores está conectada via extranet, as empresas não precisam ficar emitindo pedidos e esperar alguns dias para a confirmação dos mesmos;
- **Objetivo comum:** as empresas envolvidas convergem seus interesses em apenas num, que é satisfazer o cliente final;
- **Custo:** além de requerer menos mão de obra, as empresas economizam também no espaço físico, nas transações com papéis e principalmente no tempo;
- **Fidelidade:** trabalhando em tempo real, a empresa pode responder de forma mais eficaz às exigências de seus clientes, aumentando a chance de tornar o cliente fiel;
- **Negociação direta:** eliminação dos intermediários nas negociações entre compradores e fornecedores, trazendo economias da ordem de 30% chegando até 80%.

Apesar das vantagens obtidas e da inquestionável tendência de digitalização da economia e dos negócios, segundo [ALB 00], o *e-business* ainda oferece algumas desvantagens:

- **Congestionamento:** muita trabalho ainda deve ser feito até tornar a internet acessível ao grande público. Além disso, novas tecnologias, já em desenvolvimento, necessitam ser implantadas para tornar a internet mais veloz e, conseqüentemente, um ambiente mais eficaz para se realizar negócios;
- **Segurança:** apesar dos avanços, a internet não é totalmente segura e está sujeita a diversos problemas de invasão, roubo de informações confidenciais, entre outros;
- **Limitações:** para que o *e-business* realmente dê vantagem competitiva à empresa, é preciso que esta esteja preparada para tal. Isso quer dizer que a

empresa deverá estar munida de pessoal capacitado e muito bem treinado, provendo-os de tecnologia da informação adequada e, o mais importante, deverá integrar, primeiramente, seus processos internos e organizá-los de maneira que possam trabalhar em tempo real, sem burocracia. Deve existir também um ambiente de desenvolvimento que permita a empresa alcançar os objetivos. Além disso, os executivos deverão apoiar e difundir essa nova idéia. Tudo isso requer tempo e também dinheiro.

5.1.2 *Outsourcing*

Outsourcing é a prática de contratar uma organização externa para desenvolver um sistema, ao invés de desenvolver na sua própria sede (*in-house*) [McC 96]. As organizações que utilizam *outsourcing* podem se especializar em uma determinada área, ter mais desenvolvedores para trabalhar em um determinado projeto, e ter uma grande biblioteca de código reutilizável. A combinação destes fatores pode resultar numa significativa redução no tempo necessário para desenvolver um produto. Muitas vezes os custos de desenvolvimento também podem diminuir.

Os maiores desafios deste tipo de ambiente de desenvolvimento são a necessidade de se avaliar muito bem a real necessidade do *outsourcing* e como ele será estruturado. Os desafios estão totalmente ligados aos riscos que uma estratégia destas envolve. Entre eles, pode-se citar [McC 96]:

- Transferência da experiência e da especialidade em um determinado assunto para fora da organização;
- Perda do controle sobre os desenvolvimentos futuros;
- Deve haver um compromisso com a informação considerada confidencial;
- Perda de visibilidade de progresso e controle dos projetos.

Além disso, muitas organizações optam pelo *outsourcing* pois se sentem frustradas pelas dificuldades de gerenciar um desenvolvimento de *software* na sua sede. A visão destas organizações é que se alguém diferente construir o *software* para elas, o trabalho pode ficar bem mais fácil. Mas, na prática, pode ocorrer o contrário.

Existe uma perda de visibilidade do progresso dos projetos quando eles são desenvolvidos no outro lado da cidade ou até mesmo no outro lado do planeta. E para compensar esta perda de visibilidade é necessário um gerenciamento bastante eficaz. Como regra, *outsourcing* necessita de muito mais gerenciamento do que um desenvolvimento *in-house*, e geralmente é um grande desafio para as organizações. Mas quando os resultados são satisfatórios, percebe-se uma grande vantagem neste tipo de ambiente.

Uma das opções do *outsourcing* que vem se tornando bastante popular ao longo dos últimos anos é o *offshore outsourcing*. Organizações *offshore* são empresas que estão localizadas em algum outro país, e que oferecem custos mais baixos de desenvolvimento. Além disso, também oferecem uma qualidade que é, no mínimo, equivalente à qualidade das organizações localizadas no próprio país [McC 96].

Em suma, uma organização pode transferir todo o seu desenvolvimento para fora da empresa ou até mesmo para fora do seu país, desde que o local seja bem selecionado e todos os aspectos sejam analisados e bem planejados. Este tipo de solução requer um adequado processo de gestão.

5.1.3 Ambientes fisicamente distribuídos

Ao optar pelo *outsourcing* ou *offshore outsourcing* uma empresa não configura um ambiente de desenvolvimento de *software* fisicamente distribuído, pois a empresa externa que foi contratada pode exercer suas atividades na própria sede da empresa contratante. A distribuição do processo de desenvolvimento de *software* ocorre somente quando parte dos envolvidos no processo estão fisicamente distantes. Este talvez seja um dos mais desafiadores contextos que o atual ambiente de negócios apresenta. Muitas empresas estão distribuindo seu processo de desenvolvimento de *software* em países como Índia, Irlanda e Brasil. Muitas vezes este processo se dá dentro de um mesmo país, em regiões com incentivos fiscais ou de concentração de massa crítica em determinadas áreas. As empresas buscam vantagens competitivas em termos de custos, qualidade ou flexibilidade na área de desenvolvimento de

sistemas [PRI, 02], além de ganhos de produtividade e diluição de riscos [McC 96]. Neste caso, ao optar por instanciar um ambiente de desenvolvimento distante fisicamente da sua sede, uma organização começa a encarar diversos desafios de adaptação, diferenças culturais, planejamento do trabalho, treinamento da nova equipe, entre outros. Sendo assim, os desafios existentes no *outsourcing* ganham proporções maiores, mas o resultado pode ser muito significativo.

Em suma, desenvolver ambientes tecnológicos, modelos e ferramentas para atuar neste tipo de ambiente é um desafio cada vez mais importante, tanto para a teoria na área de engenharia de software, como para as empresas que enfrentam as dificuldades criadas por este ambiente.

5.2 Gerenciamento de riscos

Segundo [McC 96], gerenciar riscos significa basicamente identificar, tratar e eliminar fontes de riscos antes que eles se tornem uma ameaça concreta para o término de um projeto de *software*. Riscos podem ser tratados em diferentes níveis e a gerência de risco compreende algumas categorias e subcategorias, como pode ser visto na figura 9.

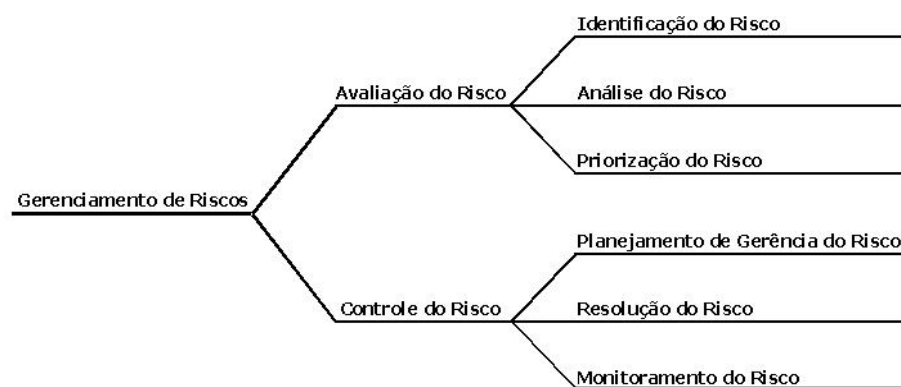


Figura 9 - Gerência de risco segundo Boehm, 1989.
Fonte: [SOM 95].

O gerenciamento de risco não é uma tarefa simples, e é tida como desejável dentro do processo de desenvolvimento de *software*. Atualmente um dos maiores desafios é a avaliação de riscos, prevendo possíveis danos que possam ser causados

no futuro. A não existência de uma gerência de riscos pode acarretar em muitos problemas indesejáveis para a empresa e para o cliente.

5.3 Capacitação de Pessoal

Capacitar pessoas é um desafio para qualquer organização. Para um *software* ser desenvolvido de maneira eficiente e eficaz, uma equipe de desenvolvimento deve estar totalmente engajada no processo, comprometida com os resultados. Mas para que isso ocorra, deve haver um investimento na capacitação da equipe, pois um dos fatores críticos de sucesso está na capacidade das pessoas seguirem os padrões definidos, desenvolvendo produtos com qualidade [PMB 00], [PRE 01].

Outro grande desafio no contexto da capacitação e gerenciamento de recursos humanos é a capacidade que uma equipe tem para se adequar a mudanças e novos cenários. Muitas vezes existe uma certa resistência em se implantar novos métodos de trabalho, e isto se reflete diretamente no produto que é desenvolvido [SCH 00].

Por estes motivos, capacitar pessoal e gerenciar corretamente os recursos humanos disponíveis envolve um bom planejamento e o desafio consiste em fazer com que isto ocorra da melhor maneira possível [PMB 00], [SCH 00].

5.4 Planejamento

Muitas organizações não possuem um planejamento adequado no que diz respeito ao desenvolvimento de *software*. Antes de se iniciar o desenvolvimento de qualquer sistema um passo obrigatório deve ser o planejamento. Planejar significa definir as estratégias que deverão conduzir o processo de desenvolvimento como um todo, ao longo do tempo. E esta etapa deve ser uma etapa preliminar a um conjunto de ciclos de projetos derivados deste processo de planejamento [PRI 02].

Muitas vezes este planejamento não ocorre, o que faz com que a empresa não tenha uma visão geral de todos os projetos que estão sendo desenvolvidos, nem dos projetos que estão por vir, e muito menos da prioridade de cada um deles. Além disso, diversas outras informações podem ser perdidas sem um planejamento e uma visão de futuro adequados.

Definir as estratégias da empresa na área de sistemas de informação, a partir de um processo formal de planejamento é um grande desafio para as organizações [AUD 00].

5.5 Padrões de Desenvolvimento de *Software*

Desenvolver *software* respeitando padrões é muito importante quando se busca qualidade, facilidades de manutenção, reutilização de código e clareza no entendimento do que está sendo desenvolvido [COR 01]. Muitas empresas não investem na padronização de suas atividades, o que acaba deixando cada integrante da equipe de desenvolvimento livre para implantar o seu próprio método de desenvolvimento. A não utilização de uma padronização, tanto em código quanto em processo e documentação, acaba por transformar um sistema num conjunto de módulos, sendo que cada módulo só é entendido pelo seu próprio autor. Além disso, a não padronização no processo de desenvolvimento de *software* compromete aspectos relacionados com qualidade e confiabilidade do projeto.

5.6 Certificação

De acordo com o que foi dito na introdução deste trabalho, as organizações têm buscado modelos de verificação e certificação do nível de maturidade do seu processo de desenvolvimento de *software*, tipo *CMM (Capability Maturity Model)*. Isto se deve à necessidade de as organizações contratantes terem um mínimo de garantia sobre a qualidade do processo utilizado pela organização ou laboratório de desenvolvimento de sistemas parceiros.

Desenvolver *software* de qualidade, com produtividade, dentro dos prazos estabelecidos e sem necessitar de mais recursos do que os alocados tem sido o grande desafio da Engenharia de *Software* [ADD 02]. A principal causa dos problemas, apontada pelos especialistas, é a falta de um processo de desenvolvimento claramente definido e efetivo. Conhecer o processo significa conhecer como os produtos e serviços são planejados, produzidos e entregues.

Neste contexto surgem as certificações de qualidade, nos seus diversos níveis. A certificação em modelos de referência demonstra a conformidade da empresa em relação a requisitos de padrões normativos nacionais e internacionais. Segundo [TEC 02], uma empresa certificada pode ter as seguintes vantagens:

- Garantia de níveis de qualidade mínimos dos fornecedores;
- Mobilização e motivação dos funcionários diante do objetivo comum de alcançar e manter a certificação de qualidade;
- Avaliação contínua da eficiência do sistema implantado, com a correção das "não-conformidades" apontadas;
- Atendimento às exigências do mercado;
- Comprovação do empenho da empresa em relação à oferta de serviços e produtos de qualidade;
- Ganho em agilidade com a maior integração entre as funções da empresa, divisão de responsabilidades e treinamento dos funcionários;
- Construção de um sistema de gestão adequado à realidade da empresa e possibilidade de saltos de produtividade.

No caso específico do CMM, no processo de desenvolvimento de *software*, se enfatiza a documentação dos processos, seguindo a premissa de que para construir ou realizar alguma melhoria do processo é preciso conhecê-lo e entendê-lo, e que a qualidade de um produto é reflexo da qualidade e gerenciamento do processo utilizado no seu desenvolvimento.

5.7 Aprendizagem Organizacional

A aprendizagem organizacional é o processo em que a base de valor e o conhecimento da organização mudam, aumentando a habilidade em resolver problemas e a capacidade de ação da organização de acordo com a demanda do ambiente [PRO 97]. A aprendizagem organizacional tem sido identificada como um elemento importante para resolver problemas nas organizações, especialmente os problemas relacionados com a grande pressão competitiva do mercado e as mudanças

de base tecnológica. O uso de técnicas criativas ([ALT 99] e [COU 96]) e o pensamento dinâmico não-linear [KAO 97] executam um papel importante na criação de um ambiente de aprendizagem. Segundo [ARG 93] e [SEN 90], nos últimos anos a aprendizagem organizacional tem atraído a atenção de gerentes e pesquisadores. Este foco na aprendizagem indica uma grande relevância da abordagem cognitiva, onde o conhecimento e as percepções dos indivíduos são vistos como elementos críticos na eficiência da organização. Atualmente, alguns pesquisadores estão desenvolvendo estudos para compreender como a aprendizagem organizacional pode ser usada no processo de planejamento de sistemas de informação [ANG 97], [BAE 98], [REP 98] e [AUD 01].

A incorporação de técnicas da aprendizagem organizacional é um dos grandes desafios do desenvolvimento de *software* atualmente, pois sua importância vem sendo destacada cada vez mais [COU 96]. Cabe às organizações fazer a correta utilização destas técnicas para obter melhores resultados no processo de desenvolvimento como um todo. E o fator crítico de sucesso para que isto aconteça está na valorização das pessoas, seus conhecimentos e experiências.

5.8 Produtividade e Motivação

Em 1954, A. Maslow disse que as pessoas eram motivadas por satisfazer suas necessidades, e estas poderiam ser organizadas em um conjunto de níveis, como mostra a figura 10. As prioridades humanas são primeiramente de satisfazer as necessidades dos níveis mais baixos, antes das necessidades mais abstratas, nos níveis superiores [SOM 95].

Para as pessoas que trabalham com desenvolvimento de *software*, pode-se assumir que os níveis mais baixos (necessidades fisiológicas e segurança) já estão satisfeitas. Entretanto, assegurar as necessidades de satisfação social, reconhecimento e realização pessoal são considerados desafios sob um ponto de vista gerencial. Sendo assim, um dos grandes desafios do desenvolvimento de *software* é motivar as pessoas de forma que isto se reflita positivamente na produtividade. Por

último, um fator adicional que deve ser considerado na motivação é o fato de que, quando uma necessidade é satisfeita, ela deixa de ser um fator motivador. Sendo assim, deve-se trabalhar continuamente buscando fatores de motivação [SCH 00].



Figura 10 - Modelo de motivação em um conjunto de níveis
Fonte: [SOM 95].

5.9 Conclusões

O processo de desenvolvimento de *software* tem avançado muito nos últimos anos, mas muitos desafios devem ser vencidos e muitas barreiras precisam ser quebradas. Com relação aos desafios, além dos citados anteriormente, identificou-se na literatura alguns outros, tais como resolução de problemas complexos, condução do processo de mudança e a reutilização no contexto do processo de *software*.

Identificou-se uma convergência na direção de dois dos desafios identificados na literatura: o planejamento no contexto do processo de desenvolvimento de *software* [REP 98], [AUD 01], e os ambientes de desenvolvimento fisicamente distribuídos [McC 96], [PRI 02]. A literatura apresenta poucas contribuições nestas duas áreas, oferecendo um espaço oportuno para pesquisas e estudos.

A evolução dos ambientes fisicamente distribuídos de desenvolvimento de *software* mostram uma nova tendência em desenvolvimento de *software* no âmbito

mundial. Até pouco tempo atrás ninguém pensava em enviar seus projetos para o outro lado do mundo da forma que tem ocorrido hoje. Muitas empresas estão optando por soluções como esta, e estão obtendo resultados que superam suas expectativas.

Esta tendência vem crescendo nos últimos anos e permite o seguinte diagnóstico: os processos (modelos e metodologias) de desenvolvimento de *software* existentes ainda não contemplam determinadas atividades que podem ser extremamente úteis ao considerar um ambiente fisicamente distribuído.

Algumas práticas deverão ser adotadas, diversas propostas de extensão dos modelos e das metodologias atuais deverão surgir, pois será necessário desenvolver modelos para tratar de problemas específicos de projetos de *software* desenvolvidos em ambientes fisicamente distribuídos, onde as equipes precisam se comunicar e trocar informações de maneira eficiente, rápida, clara e objetiva.

Além disso, se um dos desafios do desenvolvimento de *software* era justamente o planejamento dos projetos, em um contexto de ambiente fisicamente distribuído este desafio se acentua. Um dos fatores determinantes para o sucesso do desenvolvimento de *software* em uma empresa é a capacidade de implantar um processo de planejamento integrado com o processo de desenvolvimento de *software*.

6 Abordagens de Desenvolvimento de *Software*

Este capítulo apresenta um breve histórico do processo de desenvolvimento de *software*, apresentando as principais abordagens existentes (Análise Estruturada, Modelagem de Dados e Orientação a Objeto). Ao final do capítulo é feita uma comparação entre as abordagens estudadas.

6.1 Evolução das Abordagens de Desenvolvimento de *Software*

Ao longo das últimas décadas a engenharia de *software* passou por uma grande evolução, uma mudança radical em termos de metodologias / abordagens de desenvolvimento de *software*. Diversos autores deram contribuições importantes para a engenharia de *software* chegar aos dias de hoje de forma organizada e definida.

A seguir, na figura 11, pode ser vista a ordem cronológica de como toda esta evolução ocorreu ao longo do tempo:

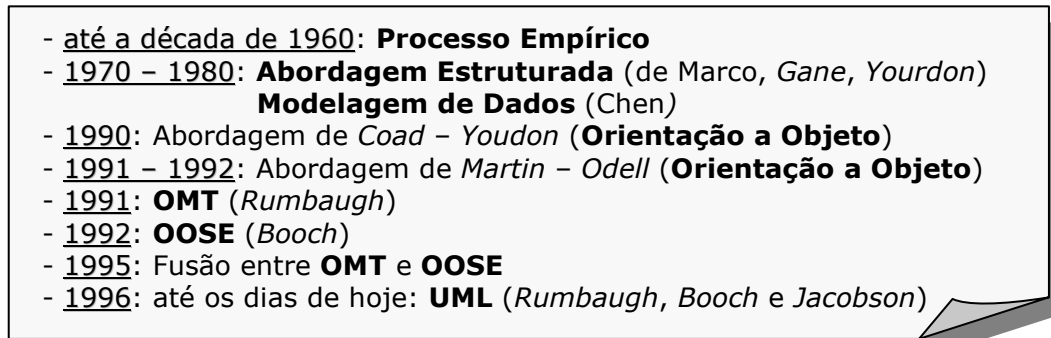
- 
- até a década de 1960: **Processo Empírico**
 - 1970 – 1980: **Abordagem Estruturada** (de Marco, Gane, Yourdon)
Modelagem de Dados (Chen)
 - 1990: Abordagem de Coad – Yourdon (**Orientação a Objeto**)
 - 1991 – 1992: Abordagem de Martin – Odell (**Orientação a Objeto**)
 - 1991: **OMT** (Rumbaugh)
 - 1992: **OOSE** (Booch)
 - 1995: Fusão entre **OMT** e **OOSE**
 - 1996: até os dias de hoje: **UML** (Rumbaugh, Booch e Jacobson)

Figura 11 - Evolução das abordagens de desenvolvimento de *software*.

Considerando as abordagens orientadas a objeto, a figura 12 traz uma visão histórica da evolução que ocorreu até o surgimento da primeira versão da UML (Linguagem Unificada de Modelagem), em 1996, e submetida para o OMG (*Object Management Group*) em janeiro de 1997.

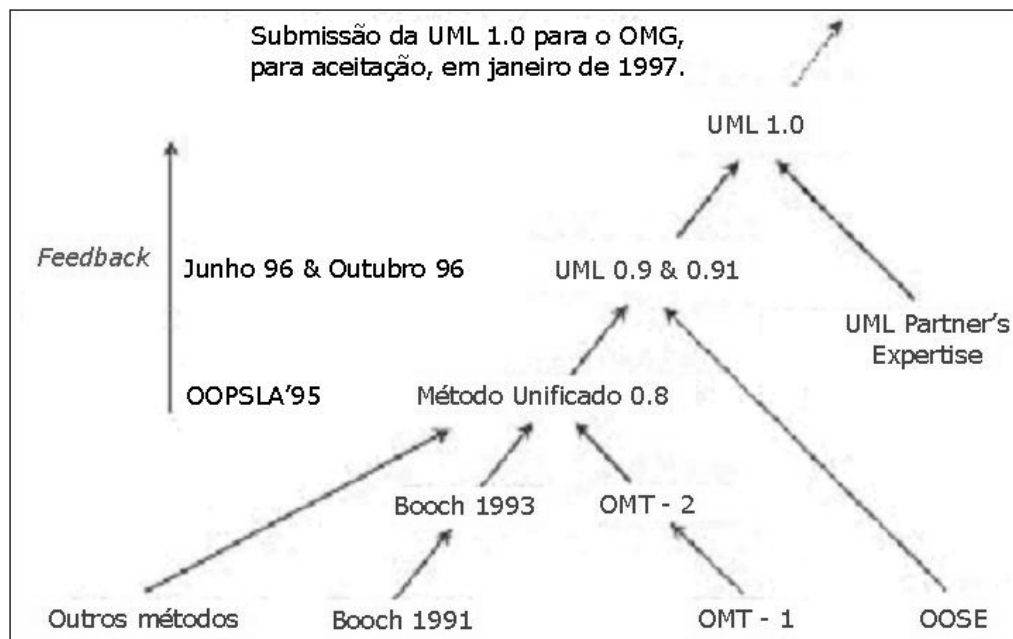


Figura 12 - Evolução da UML
 Fonte: <http://www.rational.com>

A seguir serão apresentadas as três abordagens principais surgidas nas últimas décadas. Especial ênfase será dada para a Orientação a Objeto, visto a importância desta abordagem para este processo de pesquisa.

6.2 Abordagem Estruturada

Nas primeiras décadas de uso dos computadores, o processo de desenvolvimento de *software* podia ser caracterizado como empírico, ou seja, direcionado pela prática, sem uma base conceitual ou metodológica coerente. Naquela época desenvolver *software* era um processo individual, quase artístico.

Em meados do anos 70, o crescimento da importância dos computadores nas empresas, evolução das linguagens de programação, problemas nos sistemas desenvolvidos, demandas por alguma padronização e sistemática no processo de desenvolvimento de *software* levaram ao surgimento de estudos que analisaram os problemas existentes na área de *software*. Como solução para os problemas identificados, surgiram propostas de maior rigor e método no processo de desenvolvimento. Estas primeiras abordagens mantiveram uma orientação básica centrada no fluxo de dados das aplicações (ou processos).

A partir de estudos de *De Marco* (1979), diversos autores ([GAN 79], [YOU 90], entre outros) apresentaram variantes e enriqueceram a teoria na área de engenharia de *software*, centrada basicamente em processo de desenvolvimento.

A abordagem estruturada dá ênfase na decomposição funcional, onde um sistema é encarado basicamente como o fornecedor de uma ou mais funções para o usuário final. Além disto, esta abordagem abrange algumas notações para a especificação formal de um sistema. Esta abordagem foi uma das primeiras a ser planejada para o desenvolvimento de *software* de uma maneira mais formal e cuidadosa, centrando o desenvolvimento nas fases de análise, projeto e implementação.

Existem alguns artefatos que são utilizados durante a fase de análise, tais como:

- Diagrama de Fluxo de Dados (DFDs)
- Dicionário de Dados (DD)
- Diagramas de Entidades Relacionadas (DERs)
- Diagramas de Transição de Estado (DTEs)
- Especificação de Processo

O **diagrama de fluxo de dados** fornece um meio gráfico de modelar o fluxo de dados pelo sistema. Um sistema típico requer diversos níveis de diagramas de fluxo de dados.

O segundo principal artefato para modelagem da análise estruturada é o **dicionário de dados**, que fornece a informação de texto de suporte para complementar a informação gráfica mostrada no DFD. Um DD é simplesmente um grupo organizado de definições de todos os elementos de dados no sistema sendo modelado.

O terceiro artefato da análise estruturada é o **diagrama de entidades relacionadas**, o DER. Ele enfatiza os principais objetos, ou entidades com que o sistema lida, bem como a relação entre os objetos, que normalmente correspondem, um a um, aos locais de armazenagem de dados mostrados no DFD. O DFD não informa sobre as relações entre os objetos. O DER também fornece uma visão simples

e gráfica do sistema para certos usuários que podem não se importar muito com os detalhes funcionais. Ele destaca a informação que não é óbvia no DFD.

O **diagrama de transição de estados**, ou DTE, é o quarto artefato da análise estruturada. Ele é usado para modelar o comportamento do sistema de acordo com certas entradas de dados. O sistema deve estar preparado para receber várias combinações e seqüências de entradas, às quais devem receber respostas apropriadas, e o DTE modela este comportamento.

Por último, o quinto artefato principal da análise estruturada é a **especificação de processo**. Sua finalidade é permitir que o analista de sistemas descreva rigorosa e precisamente a política representada por cada um dos processos "atômicos" de baixo nível nos diagramas de fluxo de dados de baixo nível.

Estas são as principais partes da análise estruturada, que são o caminho que levará ao projeto estruturado e logo após à programação estruturada.

O projeto estruturado aborda os detalhes de nível inferior do que foi gerado nesta fase. Na fase de projeto, são acrescentados detalhes aos modelos de análise e os diagramas de fluxo de dados são convertidos em descrições de diagramas de estruturas, representando a codificação em linguagem de programação. Novos artefatos são gerados.

Na programação estruturada os sistemas são compostos de subprogramas. Este processo de decomposição funcional é o mais tradicional fundamento da análise estruturada. A decomposição gradativa dos subprogramas leva, no seu nível mais fundamental, aos procedimentos básicos do sistema. A decomposição funcional força o programador a fixar atenção muito mais nos procedimentos do que nos dados. É justamente esse desequilíbrio de importância uma das principais fontes de críticas à análise estruturada.

Diversos problemas existentes nas abordagens estruturadas deram início à estudos que buscavam outras alternativas para a condução do processo de desenvolvimento de *software*. Entre estes problemas se destacam o fato de que os aspectos funcionais são os componentes mais instáveis dos requisitos dos sistemas, a

passagem da análise para o projeto é difícil, não há preocupação efetiva com a reutilização e geralmente estas técnicas não correspondem corretamente às mudanças no domínio do problema. Por outro lado, a vantagem principal das abordagens estruturadas é que elas representaram um significativo avanço no processo de desenvolvimento de *software*, ao sistematizarem e organizarem o processo como um todo.

6.3 Modelagem de Dados

Com o surgimento dos SGBDs, surgiu a necessidade de uma maior atenção durante a modelagem dos dados. Isto se tornou uma atividade crítica no processo de desenvolvimento de *software*. Diversos autores adotaram a modelagem de dados como sendo o próprio processo de desenvolvimento, entre eles *James Martin* em 1994. A Engenharia da Informação (EI) é um exemplo disto, centrando todo o processo de desenvolvimento de *software* na modelagem de dados. A modelagem da informação originou-se na comunidade de banco de dados e se preocupa com que a modelagem da estrutura de dados possa ser gerenciada adequadamente em um banco de dados [RUM 91].

A abordagem de Entidade-Relacionamento (ER) – *Chen* 1976, é a abordagem mais comum de modelagem da informação. A notação é de fácil entendimento e os diagramas podem ser facilmente traduzidos para uma implementação de banco de dados.

A Engenharia da Informação aplica técnicas estruturadas que envolvem toda a empresa ou uma área da empresa, não analisando projetos isoladamente. A EI envolve quatro etapas: planejamento, análise do negócio, projeto e construção. À medida que avança por estas etapas, a EI gera constantemente um repositório com conhecimentos sobre a empresa, seus projetos de sistemas e a implementação deles [MAR 96].

Um dos objetivos da EI é identificar as situações comuns tanto dos dados como dos processos e minimizar o trabalho de desenvolvimento de sistemas redundantes.

Na dimensão de dados, baseia-se na modelagem de dados, enquanto que a dimensão de processos (métodos) utiliza técnicas da abordagem estruturada. A figura 13 ilustra estas duas dimensões.

Os principais problemas da Modelagem de Dados enquanto um processo de desenvolvimento de *software* estão relacionados com a falta de técnicas e ferramentas adequadas para a modelagem dinâmica da aplicação em desenvolvimento. Em outras palavras, a dimensão de processos fica desbalanceada metodologicamente frente à dimensão de dados. Por outro lado, a grande vantagem desta abordagem está relacionada com a dimensão estática do sistema, envolvendo a modelagem de dados propriamente dita.

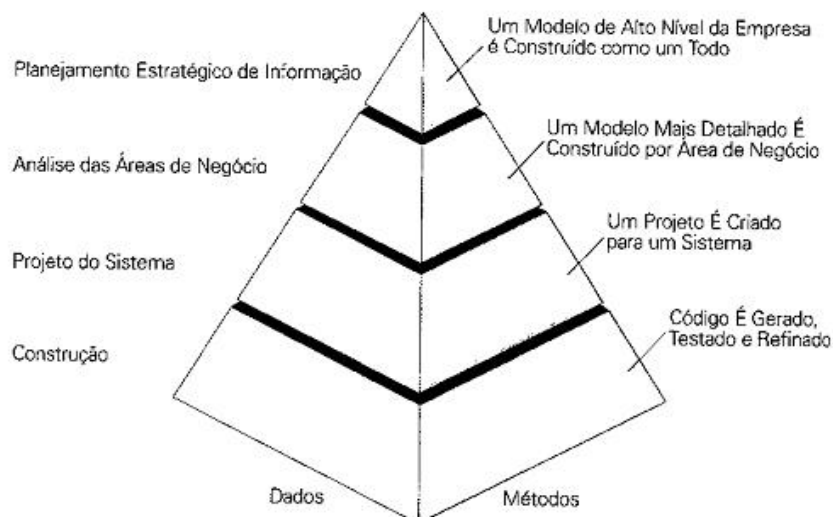


Figura 13 - Engenharia da Informação.
Fonte: [MAR 96]

6.4 Orientação a Objetos

Na última década houve uma revolução industrial no *software*, ocasionando o surgimento de ferramentas *CASE*, geradores de código, programação visual, desenvolvimento baseado na reutilização e, principalmente, de técnicas orientadas a objeto. O termo revolução na indústria do *software* foi utilizado para descrever o movimento para uma nova Era, onde o *software* deixou de ser um conjunto de pacotes monolíticos em que apenas um fabricante construía este pacote, e se

transformou em um conjunto de componentes e pacotes de vários fabricantes [MAR 96].

A Orientação a Objeto (OO) é uma abordagem de desenvolvimento baseada no conceito de que os sistemas devem ser construídos a partir de um conjunto de componentes reutilizáveis, chamados de objetos. A idéia fundamental da composição de *software* através de objetos surgiu em 1967, em uma linguagem de programação chamada *Simula67* [BAH 99]. Mas os primeiros conceitos da programação orientada a objetos foram implementados inicialmente na linguagem chamada *Smalltalk-80* [LEW 95].

Entre os motivos que influenciaram o surgimento desta abordagem, pode-se citar:

- Avanços na tecnologia de arquiteturas de computadores, suportando sofisticados ambientes de programação e interfaces homem-máquina;
- Avanços na área de linguagens de programação, como modularização, ocultamento de informação, etc;
- A crise (revolução) do *software*.

Segundo [MAR 96], as técnicas orientadas a objeto mudaram toda a indústria de *software*, em particular:

- A maneira como os pacotes de aplicativos são vendidos;
- A maneira como se utiliza os computadores;
- A maneira como se analisam os sistemas;
- A maneira como se projetam os sistemas;
- O trabalho de todos os profissionais de desenvolvimento de *software*.

6.4.1 Conceitos Básicos de Orientação a Objeto

Um objeto é uma entidade que possui um estado, exibe um comportamento bem definido e possui uma identidade única. São entidades que encapsulam dados (atributos) e um conjunto de operações associadas que manipulam esses dados. [BAH 99].

Objetos que compartilham o mesmo comportamento são ditos como pertencentes a mesma classe. Uma classe é a descrição de um grupo de objetos com propriedades similares (atributos), comportamento comum (operações), relacionamentos com outros objetos e semânticas idênticas. O conjunto de operações que um cliente de uma classe pode acessar constitui sua interface pública e a codificação da sua estrutura e de suas operações corresponde à implementação da classe.

A instância corresponde a um exemplar de uma classe. Todos os objetos são instâncias de alguma classe. Os atributos são denominados variáveis de instância e as operações são chamadas de métodos. Um método deve fazer parte da interface privada do objeto, e não da pública, pois esta especificação atende ao princípio do ocultamento de informações, que será explicado a seguir [PRE 01].

Para um objeto executar determinada ação ele deve receber uma mensagem. A mensagem corresponde a um pedido de execução de uma operação, endereçado ao objeto, e disponível em sua interface. Operações são executadas pelos objetos em resposta ao envio de mensagens [MAR 96].

O mecanismo de herança é um outro conceito que permite estender e adaptar a implementação de um determinado objeto, sem necessariamente alterar o código fonte [PRE 01], viabilizando a reutilização de classes de uma forma diferente da originalmente implementada. A classe existente que será estendida é denominada superclasse da nova classe, e a nova classe a ser criada a partir da superclasse é denominada subclasse. A subclasse herdará a interface e a implementação da superclasse.

Quando um objeto esconde seus dados de outros objetos e permite que os dados sejam acessados somente através de seus próprios métodos, caracteriza-se o ocultamento de informações (*information hiding*). O encapsulamento então oculta os detalhes de implementação interna aos usuários de um objeto. A grande importância do encapsulamento é a separação da forma de como o objeto é implementado da forma como ele se comporta [BAH 99].

6.4.2 *Análise e Projeto Orientado a Objeto*

Quando se desenvolve a análise de um sistema, criam-se modelos que devem representar uma determinada realidade [PID 97]. Um modelo pode envolver um sistema, concentrar-se numa área de negócio ou abranger uma empresa inteira. O modelo representa um aspecto da realidade e é construído de tal forma que ajude a entendê-la. Um modelo é muito mais simples do que a realidade, da mesma forma que um modelo de avião é mais simples do que um avião real, e pode ser manipulado [MAR 96].

A análise OO modela o mundo em termos de tipos de objetos e do que acontece a esses tipos de objetos, o que conseqüentemente leva a projetar/modelar e programar sistemas de uma forma orientada a objeto, para ter os benefícios que esta abordagem pode trazer.

Nas abordagens de desenvolvimento tradicionais, os modelos conceituais usados para análise diferem daqueles usados para projeto. A programação tem ainda uma terceira visão do mundo. Os analistas usam modelos de entidade-relacionamento, decomposição funcional e matrizes. Os projetistas usam diagramas de fluxo de dados, diagramas de ação e diagramas de transição de estados. E os desenvolvedores usam linguagens de programação estruturadas.

Por outro lado, nas técnicas OO, analistas, projetistas, desenvolvedores e os usuários finais usam o mesmo modelo conceitual. Todos pensam em tipos de objetos, objetos e como eles se comportam. A modelagem do sistema se torna muito mais simples [MAR 96].

Ainda segundo [MAR 96], a análise e o projeto OO têm dois aspectos importantes, ilustrados na figura 14. O primeiro aspecto se preocupa com tipos de objeto, classes, relações entre os objetos e herança, e é chamado de **Análise da Estrutura do Objeto (AEO)** e **Projeto da Estrutura do Objeto (PEO)**. O outro aspecto se preocupa com o comportamento dos objetos e com o que acontece com eles no decorrer do tempo, e é denominado **Análise do Comportamento do Objeto (ACO)** e **Projeto do Comportamento do Objeto (PCO)**.

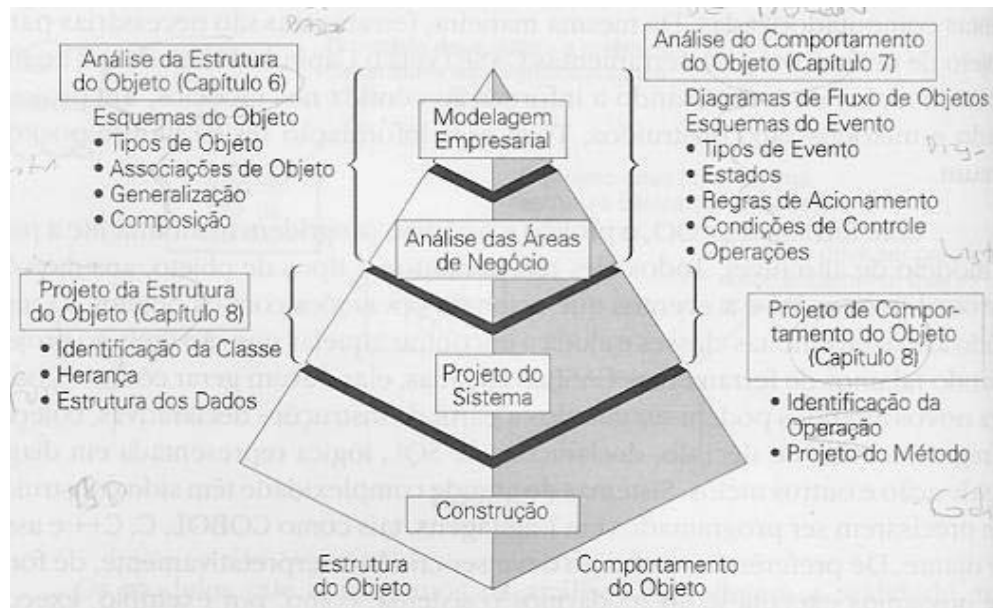


Figura 14 - Análise e Projeto Orientados a Objeto.
Fonte: [MAR 96]

Grande parte do que tem sido escrito sobre as técnicas orientadas a objeto está ancorada na programação orientada a objeto. Mas deve ser reforçado que qualquer técnica de análise e projeto orientados a objeto devem ser ensinados independentemente das linguagens de programação. A análise e o projeto OO são maneiras poderosas de pensar em sistemas complexos. A abordagem orientada a objeto tem como ponto principal modelar como a realidade é entendida pelas pessoas.

Além disso, cabe ressaltar que as técnicas orientadas a objeto mudaram a visão de como enxergar um sistema. Ao invés de se pensar em processos e na sua decomposição, passou-se a pensar em objetos e no comportamento dos mesmos.

6.4.3 A UML (Linguagem Unificada de Modelagem)

Muitos autores, entre eles [FOW 97] e [BAH 99] escreveram sobre a abordagem orientada a objeto para o desenvolvimento de *software*. As principais metodologias surgiram entre 1988 e 1992:

- Sally Shlaer e Steve Mellor escreveram dois livros em 1989 e 1991 sobre análise e projeto orientados a objeto;
- Peter Coad e Ed Yourdon também escreveram alguns livros que propuseram algumas metodologias para análise e projeto de *software* orientado a objeto entre 1991 e 1993;

- *Grady Booch* desenvolveu um trabalho considerável na empresa *Rational Software* em 1994 e 1995;
- Em 1991, *Jim Rumbaugh* liderou uma equipe em uma pesquisa nos laboratórios da *General Electric*, que resultou num livro bastante popular sobre a metodologia OMT (*Object Modeling Technique*);
- *Jim Odell* e *James Martin* contribuíram com a sua experiência para a evolução da abordagem orientada a objeto;
- Ivar Jacobson introduziu os conceitos de casos de uso em 1994 e 1995;

No ano de 1994, na época de uma das maiores conferências voltadas para a orientação a objeto, a *OOPSLA '94*, as metodologias eram bastante competitivas. Cada autor levava consigo pequenos grupos que acreditavam nas aplicações das suas idéias. Todas as metodologias eram bastante similares, mas ninguém pensava em padronizá-las.

Nesta conferência uma das grandes notícias foi a saída de *Jim Rumbaugh* da *General Electric* para se juntar a *Grady Booch* na *Rational Software*. A partir deste momento, eles declararam o fim da guerra de metodologias e no ano seguinte, no *OOPSLA '95*, eles apresentaram a primeira versão da união de suas metodologias: a versão 0.8 da metodologia unificada. Neste ano, *Ivar Jacobson* se juntou a eles na *Rational Software* e em 1996, eles ficaram conhecidos como os três amigos e passaram a trabalhar num método chamado de Linguagem Unificada de Modelagem (UML). Mas os principais nomes da comunidade de metodologias orientadas a objeto não se mostravam interessados em adotar a UML.

Neste momento o OMG (*Object Management Group*) formou uma equipe para coordenar a padronização das metodologias e em janeiro de 1997 várias organizações submeteram propostas para esta padronização. A *Rational* gerou uma versão 1.0 da UML como sendo a sua proposta. *Jim Odell* e o grupo do OMG gastaram bastante tempo trabalhando na definição de um padrão para a modelagem visual orientada a objeto. E em novembro de 1997 a UML se consolidou como uma linguagem unificada para a abordagem orientada a objeto, sendo aprovada pelo OMG.

A UML combina o melhor de Conceitos de Modelagem de Dados (Diagramas Entidade-Relacionamento), Modelagem de Negócios (Fluxo de trabalhos), Modelagem de Objetos e Modelagem de Componentes [FOW 97]. É a linguagem padrão para visualizar, especificar, construir e documentar os artefatos de um sistema intensamente baseado em *software* e pode ser usada com todos os processos, durante todo o ciclo de desenvolvimento, e com diferentes tecnologias de implementação.

A UML utiliza artefatos como casos de uso, atores, diagramas de iterações, diagrama de classes, entre outros, utilizada para:

- Mostrar a periferia de um sistema e suas maiores funções usando Casos de Uso e Atores;
- Ilustrar realizações de Casos de Uso com Diagramas de Iterações;
- Representar a estrutura estática de um sistema usando Diagramas de Classes;
- Modelar o comportamento de objetos com Diagramas de Transições de Estado;
- Revelar a arquitetura de implementação física com Diagramas de Componentes e Distribuição;
- Extender sua funcionalidade com Estereótipos.

6.4.4 *Vantagens da Abordagem Orientada a Objeto*

O uso da orientação a objetos propicia diversas vantagens. [MAR 96] e [BAH 99] destacam os seguintes aspectos:

- **Reusabilidade:** facilita a reutilização de código através dos conceitos herança, polimorfismo, encapsulamento, modularidade e coesão;
- **Extensibilidade:** para acrescentar novos recursos ao sistema basta introduzir modificações necessárias em só um lugar - a classe apropriada.
- **Aumento da qualidade:** os modelos refletem o mundo real de maneira mais aproximada e descrevem os dados de maneira mais precisa;
- **Manutenibilidade:** os sistemas são mais fáceis de entender e de manter;

- **Vantagens financeiras:** o custo de construção de *software* é reduzido;

Algumas destas vantagens são dependentes entre si. Por exemplo, o fato de proporcionar reusabilidade pode tornar o projeto mais rápido e reduzir o custo de construção de *software*.

6.4.5 Os Problemas Potenciais da Orientação a Objetos

Se por um lado esta abordagem propicia diversas vantagens, também podem ocorrer alguns problemas [MAR 96]:

- **Análise e Projeto:** a orientação a objeto exige maior concentração na análise e no projeto. Para isto, a comunidade de usuários e a administração das empresas precisam aceitar este fato.
- **Comunicação com o usuário:** os desenvolvedores precisam trabalhar em conjunto com os usuários, pois eles entendem da mecânica de seus negócios.
- **Mudança de mentalidade e de cultura:** a OO exige uma completa modificação da mentalidade do indivíduo e uma mudança na cultura de desenvolvimento dos departamentos de sistemas.
- **Trabalho em equipe:** as técnicas de OO não garantem a construção dos sistemas adequados. Todos os desenvolvedores, usuários e encarregados da manutenção precisam trabalhar em conjunto.

6.5 Abordagem Estruturada x Orientação a Objetos

A principal diferença entre a abordagem estruturada e o enfoque da orientação a objetos está na forma pela qual dados e procedimentos se intercomunicam.

Na abordagem estruturada, a principal ênfase é dada aos procedimentos. Estes são implementados em blocos estruturados e a comunicação entre os mesmos se dá pela passagem de dados. Os dados são processados dentro de blocos e migram de um para outro. Um programa estruturado, quando em execução, é caracterizado pelo acionamento de procedimentos cuja tarefa é a manipulação dos dados. Esta abordagem de desenvolvimento é baseada no conceito de que um sistema deve ser

dividido em duas partes: os dados e a funcionalidade. Assim, desenvolvem-se aplicações onde os dados ficam separados do comportamento tanto no projeto quanto na implementação do sistema [RUM 91].

Na orientação a objetos, dados e procedimentos fazem parte de um só elemento básico (objeto ou classe). Esses elementos básicos, ao estabelecer comunicação entre si, caracterizam a execução do programa. Assim, ao contrário da filosofia estruturada, onde dados e procedimentos são entidades dissociadas, na abordagem orientada a objeto os dados e procedimentos estão encapsulados em um só elemento. Ao invés de definir sistemas como duas partes separadas, se passa a conhecer o sistema como um conjunto de objetos interativos, que desempenham ações e armazenam informações [RUM 91].

Em suma, como pode ser visto na figura 15, nas técnicas de desenvolvimento estruturado, os modelos conceituais usados para análise diferem daqueles usados para projeto. Por outro lado, nas técnicas orientadas a objeto, analistas, projetistas, desenvolvedores e, de forma particularmente importante, os usuários finais, usam o mesmo modelo conceitual [MAR 96].

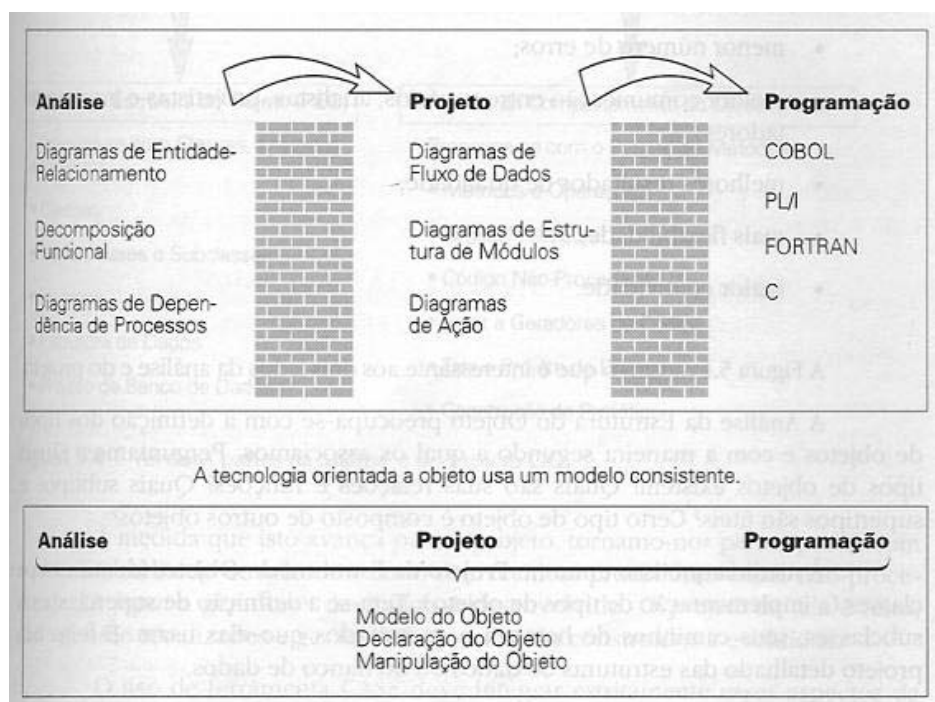


Figura 15 - Modelos conceituais em cada abordagem.
Fonte: [MAR 96]

A transição da análise para o projeto na OO é bastante natural, tornando difícil especificar onde a análise termina e onde inicia o projeto.

6.5.1 *Vantagens da OO em relação à Abordagem Estruturada*

Durante a última década, diversos autores ([MAR 96], [ROY 98], [PRE 01], entre outros) têm destacado a vantagem da abordagem orientada a objeto sobre a abordagem estruturada. Tendo como exemplo a figura 15, o emprego de um único modelo conceitual com uma ferramenta *CASE* integrada para esse modelo resulta em algumas vantagens, tais como:

- maior produtividade;
- menor número de erros;
- melhor comunicação entre usuários, analistas, projetistas e desenvolvedores;
- melhores resultados de qualidade;
- maior flexibilidade;
- maior criatividade.

A idéia original que norteou o desenvolvimento da orientação a objetos foi a de evitar o desperdício de esforço em se programar várias vezes a mesma coisa, ou seja, “reinventar a roda” toda vez que se faz um *software*. Para resolver este problema, na abordagem orientada a objeto, deve-se pensar sempre na reutilização de códigos para que o esforço do programador esteja concentrado na parte verdadeiramente importante do programa. Assim, por exemplo, códigos referentes à interface podem ser reaproveitados em diversos programas, sem dificuldades de alteração. Esta característica garante alto índice de reutilização.

CONCLUSÕES

A guisa de conclusão, o método de pesquisa utilizado neste trabalho foi basicamente a revisão bibliográfica, onde buscou-se consolidar os conhecimentos relativos aos principais problemas, desafios, ciclos de vida e abordagens do processo de desenvolvimento de software.

Identificou-se uma grande convergência na literatura sobre os principais problemas na área de desenvolvimento de *software*. Como contribuição deste estudo, buscou-se categorizar os principais problemas. Com relação aos desafios, buscou-se convergir para os dois mais significativos no contexto desta pesquisa, quais sejam o planejamento e o ambiente de desenvolvimento fisicamente distribuído. Com relação às abordagens e ciclos de vida, o objetivo foi de mapear os principais modelos existentes e tecer algumas considerações e análises críticas sobre eles.

Na continuidade deste processo de pesquisa pretende-se evoluir o estudo em direção ao projeto de dissertação de mestrado em curso. Pretende-se, no Trabalho Individual II:

- Aprofundar os estudos nos principais modelos de processo de desenvolvimento de *software* baseados na abordagem orientada a objeto, buscando identificar o estado da arte na área. Entre estes modelos destacam-se o RUP (*Rational Unified Process*), o MSF (*Microsoft Solution Framework*), o XP (*Extreme Programming*), entre outros;

- Aprofundar os estudos na área de planejamento de sistemas de informação, considerando o planejamento como uma etapa preliminar a um conjunto de ciclos de projetos de desenvolvimento de *software*;

- Aprofundar os estudos com relação ao impacto de ambientes fisicamente distribuídos no processo de desenvolvimento de *software*.

REFERÊNCIAS BIBLIOGRÁFICAS

- [ADD 02] <http://www.addtech.com.br/Servicos/CMM/cmmobjetivo.htm>
- [ALB 00] ALBERTIN, A. L. **Comércio Eletrônico: modelos, aspectos e contribuições de sua aplicação**. 2. Ed. – São Paulo: Atlas, 2000.
- [ALT 99] ALTIER, W. **Effective processes for problem solving e decision-making**. New York, Oxford Press, 1999.
- [ANG 97] ANG, K., THONG, J e YAP, C. IT implementation through the lens of OL: a case study of insuror. **Proceedings of ICIS**, Atlanta, 1997.
- [ARG 93] ARGYRIS, C. **On organizational learning**. Oxford, Blackwell, 1993.
- [AUD 00] AUDY, J. e BECKER, J. **As diferentes visões do processo decisório: do modelo racional ao modelo político e o impacto da subjetividade**. Revista Análise 11(2), Porto Alegre, Edipucrs, 2000.
- [AUD 01] AUDY, J. L. N. Modelo de Planejamento Estrategico de Sistemas de Informação: Contribuições do processo decisório e da aprendizagem organizacional. Tese de Doutorado, PPGA – UFRGS, POA, Brasil, 2001.
- [BAE 98] BAETS, W. **Organizational Learning and Knowledge Technologies in a dynamic environment**. Dordrecht, The Netherlands, Kluwer Academics Publishers, 1998.
- [BAH 99] BAHRAMI, A. **Object Oriented Systems Development – using the unified modeling language**. McGraw-Hill, 1999.
- [BOA 93] BOAR, B. **The Art of Strategic Planning for Information Technology**. New York, John Wiley and Sons, 1993.
- [BOE 01] BOEHM, B. Spiral Development: Experience, Principles and Refinements. **Spiral Development Workshop**, 2000. Disponível em

<http://www.sei.cmu.edu/publications/documents/00.reports/00sr008.html>.

- [BOE 91] BOEHM, B. **Software Risk Management: principles and practices**. IEEE Transactions Software Engineering, vol. 18, no. 1, jan., 1991.
- [CAN 98] CANTOR, M. **Object-oriented management with UML**, 1998.
- [CAR 01] CARVALHO, A. M. B. R., CHIOSSI, T. C. S. **Introdução à Engenharia de Software**. Editora Unicamp, 2001.
- [COA 92] COAD, P. e YOURDON, E. **Análise Baseada em Objetos**. Editora Campus, Série Yourdon Press 1992.
- [COR 01] CORTES, M. L., CHIOSSI, T. S. **Modelos de Qualidade de Software**, Editora da Unicamp, 2001.
- [COU 96] COUGER, J. **Creativity and innovation in IS organizations**. Danver, MA, Boyd and Fraser Publishing, 1996.
- [FOW 97] FOWLER, M., SCOTT, K. **UML distilled: applying the standard object modeling language**. 2.ed., 1997.
- [GAN 79] GANE, C. e SARSON, T. **Análise Estruturada de Sistemas**. Livros Técnicos e Científicos Editora Sa. 1979.
- [GLA 98] GLASS, R. L. **Software Runaways – Lessons Learned from Massive Software Project Failures**. Prentice Hall, PTR, NJ, 1998.
- [GOT 97] GOTTSCHALK, P. Strategic information system planning: the implementation challenge. **Proceedings of AIS 97**, Indianapolis, USA, 1997.
- [GOT 98] GOTTSCHALK, P. Content characteristics of formal information technology strategy as implementation predictors. Thesis (Ph.D.) Henley Management College, Brunel University, 1998.
- [GRI 00] GRIMSON, J. B., KUGLER, H. J. Software Needs Engineering – a position paper. **Proceedings of the 22nd International Conference on Software Engineering**. Limerick, Ireland, 2000.
- [IEE 93] IEEE **Standards Collection: Software Engineering**, IEEE Standard

610.12 – 1990, IEEE, 1993.

- [KAO 97] KAO, J. **Jammimg**. Rio de Janeiro, Campus, 1997.
- [KEA 97] KEARNS, G. e LEDERER, A. Alignment of IS plans with business plan: the impact on competitive advantage. **Proceedings of AIS 97**, Indianapolis, USA, 1997.
- [KRU 01] KRUCHTEN, P. From Waterfall to Iterative Lifecycle – A tough transition for projects managers. **Rational Software White Paper**, 2001.
- [KIN 88] KING, W.R. **How effective is your IS planning?**. Long Range Planning (21:2), 1988.
- [LAR 00] LARMAN, C. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos**, 2000.
- [LAY 00] LAYZELL, P., BRERETON, O. P., FRENCH, A. Supporting Collaboration in Distributed Software Engineering Team. **Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00)**. 2000.
- [LED 88] LEDERER, A. L. e SETHI, V. **The implementation of strategic information system planning methodologies**. MIS Quarterly (12:3), September 1988.
- [LED 96] LEDERER, A. L. e SALMEIA, H. Toward a theory of strategic information system planning. **Journal of Strategic Information System**, 1996.
- [LEF 01] LEFFINGWELL, D. **Features, Use Cases, Requirements, Oh My!** Rational Software White Paper, 2001. Disponível em <http://www.rational.com/products/whitepapers/featucreqom.pdf>.
- [LEW 95] LEWIS, T. et al. **Object-Oriented Application Frameworks**. Prentice Hall, 1995.
- [MAC 90] MACRO, A. **Software Engineering – Concepts and Managements**. Prentice Hall, 1990.
- [MAR 96] MARTIN, J. e O'DELL, J. **Análise e Projeto Orientados a Objeto**. Makron Books, 1996.

- [MAR 98] MARTIN, J. e O'DELL, J. **Object Oriented Methods: A Foundation**. Prentice-Hall, NJ, 1998.
- [McC 96] McCONNEL, S. **Rapid Development**. Microsoft Press, 1996.
- [NAU 76] NAUR, P., RANDELL, B., BUXTON, J. N. Software Engineering – Concepts and Techniques. **Proceedings of the NATO Conferences**, 1976.
- [PET 01] PETERS, J. F., PEDRYCZ, W. **Engenharia de Software, Teoria e Prática**, Brasil, Editora Campus, 2001
- [PID 97] PIDD, M. et al. **Tools for Thinking: Modelling in Management Science**, 1997.
- [PMB 00] PMBOK Guide. **A Guide to the Project Management Body of Knowledge**, PMI, Pennsylvania, US, 2000.
- [POR 95] PORTER, M. E. e MILLAR, V. E. How Information gives you competitive advantage. **Harvard Business Review**, Boston, Jul/1995.
- [PRE 95] PRESSMAN, R. S. **Engenharia de Software**. Makron Books, 1995. Cap. 1, 7, 8 e 12, 1995.
- [PRE 01] _____. **Software Engineering. A Practitioner's Approach**. Fifth Edit, 2001.
- [PRI 02] PRIKLADNICKI, R., PERES, F., AUDY, J., MÓRA, M. C., PERDIGOTO, A. Requirements specification model in a software development process inside a physically distributed environment. **Proceedings of ICEIS 2002**, Ciudad Real, Spain, 2002.
- [PRO 97] PROBST, G. e BUCHEL, B. **Organizational learning**. London, Prentice Hall, 1997.
- [REI 96] REICH, B. e BENBASAT, I. **Measuring the linkage between business and information technology objectives**. MIS Quarterly, March 1996.
- [REP 98] REPONEN, T. **The Role of Learning in Information System Planning and Implementation**. In: GALLIERS, H. e BAETS, R. Information Technology and Organizational Transformation. Chichester,

- England, John Wiley and Sons, 1998.
- [ROY 98] ROYCE, W. **Software Project Management – a Unified Framework**. Addison-Wesley, 1998.
- [RUM 91] RUMBAUGH, J. et al. **Modelagem e Projeto Baseados em Objetos**. Editora Campus, 1991.
- [SCH 99] SCHULMEYER, G. G. e McMANUS, J. I. **Handbook of Software Quality Assurance**. 3rd edition. 1999.
- [SCH 00] SCHWALBE, K. **Information Technology Project Management**. Cambridge, 2000.
- [SEI 02] <http://www.sei.cmu.edu/>
- [SEN 90] SENGE, P. M. **A Quinta Disciplina**. São Paulo, Best Seller, 1990.
- [SER 00] SERVICE, R. e BOOCKHOLDT, J. Employing information systems for competitive advantage. **Proceedings of AMCIS**, Long Beach, CA, 2000.
- [SOM 95] SOMMERVILLE, Y. **Software Engineering**. Fifth Edition. Addison-Wesley, 1995.
- [SPR 99] SPRAGUE, R.H. e McNURLIN, B.C. **Information Systems Management in Practice**. Canadá, Prentice Hall, 1999.
- [STE 00] STEIN, Wolfgang, **There's no business like e-business**. Saarbrücken - Alemanha, Scheer Magazine, V 9, janeiro 2000.
- [TEC 02] <http://www.tecpar.br/tecpar/brasil/home/certificacao.htm>
- [TEI 79] TEIXEIRA, S. R. P. **Engenharia de Software: Experiência e Recomendações**. Editora Edgard Blücher LTDA, 1979.
- [VAL 01] VALACICH, J., GEORGE, J. e HOFFER, J. **Essentials of Systems Analysis e Design**. Prentice-Hall, NJ, 2001.
- [VEN 97] VENKATRAMAN, N. Beyond Outsourcing: Managing IT Resources as a Value Center. **Sloan Management Review**, spring 1997.
- [YOU 90] YOURDON, E. **Análise Estruturada Moderna**. Editora Campus, 1990.