



CENTRO UNIVERSITÁRIO DO TRIÂNGULO
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DA COMPLEXIDADE DE SOFTWARE BASEADO EM OBJETOS

Lilian Rodrigues Medeiros

Uberlândia, dezembro de



CENTRO UNIVERSITÁRIO DO TRIÂNGULO
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DA COMPLEXIDADE DE SOFTWARE BASEADO EM OBJETOS

Lilian Rodrigues Medeiros

Monografia apresentada ao Curso de Ciência da Computação do Centro Universitário do Triângulo, como requisito básico à obtenção do grau de Bacharel em Ciência da Computação, sob a orientação do Prof. Msc. Fabian Martins da Silva.

**Uberlândia, dezembro de
2000**

ANÁLISE DA COMPLEXIDADE DE SOFTWARE BASEADO EM OBJETOS

Lilian Rodrigues Medeiros

Monografia apresentada ao Curso de Ciência da Computação
do Centro Universitário do Triângulo, como requisito básico à
obtenção do grau de Bacharel em Ciência da Computação.

Fabian Martins da Silva,Msc. (Orientador)	
	Marcos Ferreira de Rezende,Msc.
	(Coordenador de Curso)
Cleudair Nery Júnior,Msc. (Avaliador)	Ronaldo Castro de Oliveira (Avaliador)

**Uberlândia, dezembro de
2000**

DEDICATÓRIA

Dedico este trabalho à minha família pelo incentivo e amor e, em especial à minha mãe Laci, pela pessoa linda e grandiosa que é e, sobretudo, uma mãe maravilhosa.

“ Há duas formas para viver a vida.

Uma é acreditar que não existe milagre.

E a outra é acreditar que todas as coisas são um
milagre.”

Albert Einstein

AGRADECIMENTOS

Agradeço àqueles que me inspiraram ao longo destes anos de educação formal. Em especial à minha família pela força e carinho.

Resumo

Com a globalização e a competição entre as empresas, surgiu a necessidade de sistemas flexíveis e de maneira menos complexa, que satisfaçam o usuário através de soluções simples e funcionem com corretitude. E como as aplicações atuais são cada vez mais complexas e exigem maior potencial de *hardware* é preciso que se busque uma solução para a redução de custo e tempo de desenvolvimento. E o paradigma de Orientação a Objeto promete ser a solução para a diminuição da complexidade ao auxiliar na organização da mesma e introduzir novos conceitos como objetos, classes, reutilização de código,

abstrações etc.; principalmente porque as abstrações facilitam o entendimento de sistemas complexos, já que, estes, envolvem sistemas simples. Então, o primeiro passo é a modularização do sistema, particionando-o em pequenos módulos com refinamentos independentes, seguido de medidas de análise da complexidade que apontem para a necessidade de melhorias no projeto. Assim, é importante determinar notações, métricas e diretrizes que contribuam com a redução da complexidade do sistema ainda na fase inicial de projeto, possibilitando, então a construção de sistemas menos complexos e mais flexíveis. Este trabalho procura mostrar diretrizes e medidas que podem ser tomadas tanto nas fases iniciais do processo de desenvolvimento quanto na fase de codificação do *software*, para que possa ser desenvolvido um bom sistema. Através dos tópicos abordados torna-se possível identificar as características que tornam o software mais complexo e como esta complexidade pode ser reduzida.

Sumário



Introdução.....	1
1.1. Objetivos.....	3
1.2. Descrição dos Capítulos.....	4
2. Definindo a Complexidade do Software.....	5
2.1. Introdução.....	5
2.2. Complexidade de Modelos Baseados em Processos.....	6
2.2.1. Diagramas de Fluxo de Dados.....	7
2.2.1.1 Nívelando um Diagrama de Fluxo de Dados.....	8
2.2.1.2 Os Erros Comuns.....	8
2.2.1.3 Estratégias de Refinamento do DFD.....	9
2.2.2. Diagrama Entidade Relacionamento.....	9
2.2.2.1 Refinamento do DER.....	10
2.3. Projeto de Software.....	11
2.4. Documentação do Projeto.....	13
2.5. Componentes de Programa e Interface.....	13
2.5.1. Modularização do Projeto.....	13
2.5.2. Coesão.....	14
2.5.2.1. Comparação entre níveis de Coesão.....	15
2.5.2.2. Coesões Aceitáveis.....	16
2.5.2.3. Coesões Não Aceitáveis.....	16
2.5.3. Acoplamento.....	17
2.5.4. Perda de Acoplamento.....	18
2.5.5. Aumentar a Qualidade.....	19
2.5.6. Elevar a Manutenibilidade.....	19
2.5.7. Reutilização.....	19
2.5.7.1. Níveis de Reutilização.....	20
2.6. Complexidade de Código.....	20

2.6.1.	Métricas de Qualidade do Software.....	20
2.6.1.1.	Métrica de Halstead.....	21
2.6.1.2.	Complexidade Ciclomática de McCabe.....	22
2.6.2.	Métricas de Produtividade e Qualidade do Software.....	27
2.6.3.	Métricas Orientadas ao Tamanho.....	27
2.6.4.	Métricas Orientadas à Função.....	27
	Gerenciamento do Software.....	27
2.6.6.	Construção de Rotinas.....	28
2.6.6.1.	Decomposição Top- Down.....	31
2.6.6.2.	Decomposição Bottom- Up.....	31
2.6.6.3.	Decomposição Top- Down vs Bottom-Up.....	32
2.6.7.	Fundamentos do Layout.....	32
2.6.8.	Ferramentas de Programação.....	33
2.6.8.1.	Editores.....	34
2.6.8.2.	Browser.....	34
2.7.	Analisando a Qualidade do Código.....	34
2.8.	Documentação de Código.....	35
2.8.1.	Estilo de Programação como Documentação.....	35
2.8.2.	Comentar ou Não Comentar.....	36
2.9.	Conclusão.....	37
3.	Complexidade de Modelos Baseados em Objetos.....	38
3.1.	Introdução.....	38
3.2.	Conceitos Básicos sobre a Orientação a Objetos.....	39
3.2.1.	Definição de Objeto.....	39
3.2.2.	Classes, Instância e Herança.....	41
3.2.3.	Identificação de Objetos.....	41
3.2.4.	Definição de Operações.....	43
3.2.5.	Identidade do Objeto.....	43

3.2.6.	Mensagens.....	44
3.2.6.1.	Os papéis dos Objetos nas Mensagens.....	44
3.2.6.2.	Tipos de Mensagens que um Objeto pode receber.....	45
3.2.7.	Abstração.....	45
3.2.8.	Encapsulamento.....	46
3.2.9.	Retenção de Estado.....	47
3.2.10.	Herança.....	47
3.2.11.	Polimorfismo.....	48
3.2.12.	Genericidade.....	48
3.2.13.	Acoplamento.....	48
3.2.14.	Coesão Orientada a Objeto.....	49
3.3.	Análise Orientada a Objeto.....	50
3.4.	Modelagem Orientada a Objeto.....	51
3.5.	Uma Notação para OOD.....	52
3.5.1.	Modelos e Visões.....	52
3.5.2.	Modelo Lógico vs Modelo Físico.....	53
3.5.3.	Semânticas Estáticas vs Dinâmicas.....	53
3.5.4.	Diagramas de Classe.....	54
3.5.5.	Categorias das Classes.....	55
3.5.6.	Diagrama de Transição de Estados.....	57
3.5.7.	Diagramas de Objetos.....	58
3.5.8.	Diagramas de Interação.....	60
3.5.9.	Diagramas de Módulos.....	60
3.5.10.	Diagramas de Processos.....	61
3.6.	Bibliotecas de Componentes Baseados em Objetos.....	62
3.7.	Critérios Adicionais.....	62
3.8.	Modificações e Benefícios de OOD.....	63
3.9.	Métricas de Complexidade de Software Baseado em Objetos	64

3.9.1.	Características das Métricas.....	65
3.9.2.	Tipos de Métricas.....	65
3.9.3.	Métricas Orientadas a Objetos.....	66
3.9.4.	Descrição das Métricas.....	66
3.9.4.1.	Métodos Ponderados por Classe (MPC).....	66
3.9.4.2.	Profundidade da Árvore de Herança (PAH).....	67
3.9.4.3.	Número de Filhos (NF).....	67
3.9.4.4.	Acoplamento entre Objetos de Classes (AEO).....	67
3.9.4.5.	Resposta para a Classe (RPC).....	67
3.9.4.6.	Falta de Coesão nos Métodos (FCM).....	68
3.9.4.7.	Nomeando a Categoria (NC).....	68
3.9.5.	Métricas do Nível do Sistema.....	69
3.9.6.	Complexidade da Associação.....	69
3.9.7.	Proposta de Implementação.....	70
3.10.	Conclusão.....	71
4.	Estratégias para Redução da Complexidade.....	72
4.1.	Introdução.....	72
4.2.	Diretrizes para Redução da Complexidade de Software Orientado a Objeto	72
4.2.1.	Métricas Aplicadas ao Projeto Orientado a Objeto.....	73
4.2.1.1.	Métricas Intermodulares (para complexidade do projeto do sistema)	73
4.2.1.2.	Métricas Intramodulares (complexidade semântica).....	74
4.2.1.3.	Métricas Intramodulares (complexidade procedural).....	74
4.2.2.	Complexidade de Relacionamentos.....	75
	Diretrizes para o Projeto OO.....	75
4.2.3.1.	No Projeto de Arquitetura.....	76
4.2.3.2.	No projeto Detalhado.....	77
4.2.4.	Eliminação da Herança Múltipla.....	78
4.2.5.	Eliminação de Heranças Altas e Longas.....	78

4.2.6.	Criando Associações Reutilizáveis.....	78
4.2.7.	Empacotamento de Classes em Módulos.....	79
4.3.	Design Patterns.....	79
4.3.1.	Padrões de Projeto em Smalltalk MVC:.....	81
4.3.2.	Descrições de Padrões de Projeto.....	82
4.3.2.1.	Abstract Data Type (Classe).....	82
4.3.2.2.	Abstract Factory.....	83
4.3.2.3.	Adapter.....	83
4.3.2.4.	Blackboard.....	84
4.3.2.5.	Bridge.....	84
4.3.2.6.	Broker.....	84
4.3.2.7.	Builder.....	84
4.3.2.8.	Bureaucracy.....	85
4.3.2.9.	Responsabilidade da Cadeia.....	85
4.3.2.10.	Chamada a Procedure Remoto.....	85
4.3.2.11.	Command.....	85
4.3.2.12.	Composite.....	86
4.3.2.13.	Concorrência.....	86
4.3.2.14.	Controle.....	86
4.3.2.15.	Convenience Patterns.....	86
4.3.2.16.	Data Management.....	87
4.3.2.17.	Decorator.....	87
4.3.2.18.	Decoupling.....	87
4.3.2.19.	Estado.....	87
4.3.2.20.	Evento Baseado na Integração.....	88
4.3.2.21.	Facade.....	88
4.3.2.22.	Facet.....	88
4.3.2.23.	Flyweight.....	88

4.3.2.24.	Framework.....	88
4.3.2.25.	Gerenciamento da Variável.....	89
4.3.2.26.	Integração.....	89
4.3.2.27.	Iterator.....	89
4.3.2.28.	Máquinas Virtuais.....	89
4.3.2.29.	Mediator.....	90
4.3.2.30.	Memento.....	90
4.3.2.31.	Mestre / Escravo.....	90
4.3.2.32.	Método Factory.....	91
4.3.2.33.	Método Template.....	91
4.3.2.34.	Módulo.....	91
4.3.2.35.	Objeto Nulo (Stub).....	91
4.3.2.36.	Pipeline (Pipes e Filtros).....	91
4.3.2.37.	Propagator.....	92
4.3.2.38.	Observer.....	92
4.3.2.39.	Protótipo.....	92
4.3.2.40.	Proxy.....	92
4.3.2.41.	Recoverable Distributor.....	93
4.3.2.42.	Singleton.....	93
4.3.2.43.	Strategy.....	93
4.3.2.44.	Superclasse.....	93
4.3.2.45.	Visitor.....	94
4.3.3.	Propagation Patterns.....	94
4.3.4.	Patterns para Adaptive Programming (AP).....	95
4.3.4.1.	Adaptive Dynamic Subclassing.....	95
4.3.4.2.	Adaptive Builder.....	96
4.3.4.3.	Classe Diagrama e Classe Dicionário.....	96
4.3.4.4.	Estrutura Shy Object.....	96

4.3.4.5.Adaptive Interpreter.....	97
4.3.4.6.Adaptive Visitor.....	97
4.4. Conclusão.....	99
5. Conclusão.....	100
6. Referências Bibliográficas.....	102



lista de figuras



Figura 2.1- Diagrama de Fluxo de Dados.....	7
Figura 2.2 – Diagrama Entidade Relacionamento.....	9
Figura 2.3 –Exemplo de DER complexo diversidade de relacionamentos entre entidades.....	11
Figura 2.4- Exemplo de DER complexo com muitas regras entre entidades	11
Figura 2.5- Complexidade do gráfico de Controle [PRESSMAN95].	23
Figura 2.6- Complexidade Ciclométrica [PRESSMAN95].....	25
Figura 2.7- Exemplo em C	26

Figura 2.8- Grafo do Fluxo de Controle.....	26
Figura 2.9- Gerenciamento do Software Relacionado com sua Construção [MCCONNELL93].....	27
Figura 2.10- Atividades para construção da rotina [MCCONNELL93].	29
Figura 3.1- Diagrama de Classe.....	54
Figura 3.2- Ícone Categoria de Classes [BOOCH94].....	55
Figura 3.3- Diagrama de Transição de Estado para PRODUTO-VENDÁVEL:: status atual-estoque.[PAGE80].....	58
Figura 3.4- Diagrama de Objetos.....	59
Figura 3.5- Notação Básica do Diagrama de Módulos [PRESSMAN95]	61
Figura 3.6- Notação Básica do Diagrama de Processos [PRESSMAN95]	61
Figura 3.7 –Complexidade da Associação [KOLEWE93].....	69
Figura 4.1- Escalas de Acoplamento [DIAS97].....	75
Figura 4.2- Desenvolvimento das Fases do Projeto [DIAS97]...	75
Figura 4.3- Etapas da Fase do Projeto de Arquitetura [DIAS97].	76
Figura 4.4- Padrões de Projeto compreendem um problema e uma solução que resolvam as forças contextuais em maneiras que as moldem para benefícios, conseqüências e outros padrões.....	81
Figura 4.5 - O padrão é uma solução comum que foi reinventada em muitos contextos.....	81



1. Introdução

A área de Engenharia de *Software* vem mudando bastante ao longo dos últimos vinte anos em função da natureza e complexidade das aplicações desenvolvidas. Nos anos setenta, os aplicativos utilizavam um único processador, produziam saídas alfanuméricas e recebiam entradas de forma linear. Por muitos anos, até a década de oitenta, equipes de desenvolvimento de software seguiam o modelo de desenvolvimento hierárquico (requisitos, especificações, projeto, implementação e testes). A maior parte das arquiteturas de *software* era baseada em transformações, onde a entrada era transformada para a saída, ou em transações, onde uma entrada baseada em linhas de comando determinava a seleção da função apropriada.

As aplicações atuais são mais complexas, com interface gráfica e arquitetura cliente servidor; rodando em um ou mais processadores, em sistemas operacionais distintos, em máquinas remotas. Estes sistemas são baseados em camadas de suporte para gerenciamento de janelas, redes, mensagens, segurança e dados e, como consequência, as abordagens tradicionais de desenvolvimento de *software* não funcionam bem para boa parte das aplicações atuais. Mas apesar das mudanças, alguns conceitos permanecem como, por exemplo, a abstração e métodos de análise e projeto, a prototipação de interfaces, a modularidade, a análise de ciclo de vida, o reuso, as métricas e os ambientes integrados de desenvolvimento.

O aumento da complexidade dos sistemas de computação tem causado grandes problemas principalmente nas fases de projeto e implementação, refletindo em custos e tempo de desenvolvimento. Em uma tentativa de amenizar o problema, uma abordagem chamada *hardware- software co-design* tem sido largamente estudada [BOOCH94]. Abordagem esta dirigida para o desenvolvimento conjunto de partes de *hardware* e *software* de sistemas digitais, melhorando a integração das mesmas e reduzindo custo e tempo de projeto, devido ao uso de metodologias e ferramentas que permitam um tratamento mais abstrato e um melhor conhecimento do sistema, antes mesmo de definir as partições de *hardware* e *software*.

Embora não seja novo, o interesse em *co-design* foi renovado a partir do começo desta década, devido ao desenvolvimento de técnicas, como a síntese de alto nível, que põe os projetos de *hardware* num nível de abstração mais próximo do empregado no desenvolvimento de *software*. Por outro lado, as pressões do mercado, tornam a produção de *hardware* mais adaptável e flexível.

Atualmente, se o usuário tem o perfeito conhecimento de suas necessidades, existem alguns instrumentos dos quais ele necessitará; mas muitos documentos são de difícil compreensão, tendo várias interpretações e contendo elementos de requisições essenciais.

O sistema freqüentemente muda durante seu desenvolvimento, devido principalmente à existência de muitos *softwares*, surgindo então, a necessidade do domínio do problema, principalmente em sistemas grandes que, planejados ou não, envolvem muito tempo e dificultam sua abstração. Mas tamanho não é grande virtude do sistema, pois atualmente é usual encontrar sistemas distribuídos cujo tamanho está em centenas de milhares ou milhões de linhas de código.

Freqüentemente, a complexidade tem a forma de hierarquia, onde um sistema complexo é composto de subsistema que têm outros subsistemas e então algum minúsculo nível de componentes é alcançado. Alguns destes componentes são primitivos e relativamente arbitrários e com ligações internas mais fortes que externas, envolvendo, então a separação de componentes dinâmicos.

Sistemas hierárquicos são usualmente compostos de alguns tipos diferentes de subsistemas em várias combinações; pois os sistemas mais interessantes não tem somente uma, mas diferentes hierarquias que usualmente estão presentes nos sistemas complexos.

A descoberta de abstrações e mecanismos facilita o entendimento de sistemas complexos, principalmente porque estes envolvem sistemas simples. Então, o primeiro passo para se analisar um sistema complexo é encontrar as partes da interação de estilos, com as percepções destas; pois a complexidade do sistema tem limites básicos para o desenvolvedor.

Pode-se ver duas hierarquias ortogonais do sistema: a estrutura da classe e a estrutura do objeto. Cada hierarquia está coberta com muitas classes abstratas e objetos, daí a importância da decomposição do sistema complexo em pequenas partes, onde cada uma terá refinamentos independentes.

Aplicar orientação a objeto primeiro auxilia na melhor organização da complexidade ao utilizar as propriedades da tecnologia orientada a objeto que serão discutidas posteriormente no Capítulo 3, juntamente com as medidas, diretrizes e padrões para a redução da complexidade discutidas no decorrer deste trabalho, possibilitando assim, a construção de sistemas mais produtivos e flexíveis.

1.1. Objetivos

Os objetivos a serem alcançados ao longo deste trabalho visam o aperfeiçoamento na construção dos sistemas, principalmente nesta época de competição e globalização onde a grande diferença está em fazer o software funcionar e fazê-lo corretamente. E estes são listados abaixo:

- Estudar critérios e métricas baseados em objetos para identificação da baixa ou alta complexidade;
- Identificar e discutir os benefícios associados à análise, projeto e programação OO;
- Estudar técnicas de modelagem e implementação de software OO, visando definir estratégias para redução de sua complexidade;
- Identificar aspectos relevantes dentro do OOD, buscando a redução de sua complexidade a partir desta fase do processo de desenvolvimento.

1.2. Descrição dos Capítulos

No Capítulo 2 busca-se a definição do Conceito de Complexidade de *Software*, juntamente com a Complexidade de Modelos e Código.

No Capítulo 3 tem-se uma breve definição de conceitos da Tecnologia Orientada a Objetos, seguida de métricas utilizadas para redução da complexidade de sistemas OO ainda na fase de projeto.

No Capítulo 4 busca-se identificar as estratégias para a redução da complexidade, através da descrição de padrões de projetos e diretrizes para tornar o *software* menos complexo.

2. Definindo a Complexidade do Software

2.1. Introdução

A coleção de requisições é crucial para o sucesso no desenvolvimento de sistemas. E para realizá-los com qualidade, é essencial que seu desenvolvimento seja de forma sistemática e compreensível; pois ele deverá ter o que o usuário necessita e se não o faz, provavelmente não se ajustará às suas especificações; podendo levar à crise do *software*. Esta crise se manifesta através do custo acima do planejado; da insatisfação do usuário final com o produto; do *software* com defeitos e da falta de confiabilidade no mesmo.

O fato de grande parte do ciclo de vida do *software* ser consumido em testes de manutenção tem gerado atenção para o problema de sua complexidade. Muitas medidas têm sido propostas pelos pesquisadores e estas começam analisando complexidade, informações, modificações e testes do *software*; além da manutenabilidade para seu desenvolvimento. Por isto é importante que se entenda as medidas de complexidade na teoria, para aplicá-las na prática, principalmente porque as pessoas lidam com complexidade abstraindo detalhes desnecessários.

As razões que levam à complexidade do *software* são: tamanho, número de variáveis e funções; restrições de tempo; gerenciamento de memória, concorrência e *interface* orientada a eventos. Assim, Booch [BOOCH90] identificou cinco características comuns a todos os sistemas complexos:

- Existe alguma hierarquia ;
- Os componentes básicos dependem de seu ponto de vista;
- Os componentes são mais acoplados internamente que externamente;
- Existem padrões comuns usando componentes simples, capazes de representar componentes complexos;
- Geralmente sistemas complexos são construídos a partir de sistemas básicos.

As tendências da indústria de *software* estão focadas em modelagens poderosas; modelagens e programação integradas; sistemas grandes e distribuídos e utilização da arquitetura cliente servidor, ampliando a importância do domínio da complexidade nos sistemas desenvolvidos.

2.2. Complexidade de Modelos Baseados em Processos

O processo de desenvolvimento pode ser dividido em: análise, projeto e implementação; onde a análise objetiva entender o problema como preparação para o projeto; seguido pela modelagem do sistema com conceitos do mundo real de uma forma que possa ser entendido. Daí o analista interage intensamente com o requisitante, a fim de esclarecer ambigüidades e mal entendidos.

O projeto é, essencialmente, um processo de refinamento e acréscimo de detalhes, onde o projetista irá: combinar os três modelos de análise, obter as operações sobre as classes, projetar algoritmos para implementar tais operações, otimizar o caminho de acesso aos dados, implementar controles para as interações externas, ajustar a estrutura de classes para aumentar a herança, projetar associações adequadas, determinar a representação dos objetos, e empacotar classes e associações em módulos.

O processo como um todo é bastante interativo, pois quando o projeto estiver completo, em um dado nível de abstração, ele poderá ser detalhado, acrescentando novas operações ou explicitando as existentes; identificando novas classes e revisando relacionamentos.

2.2.1. Diagramas de Fluxo de Dados

É utilizado para particionar um sistema, sendo a principal ferramenta para análise estruturada, juntamente com o dicionário de dados. É também conhecido como Gráfico de Bolha. Seus componentes são:

- Fluxo de dados que é um elemento importante no DFD, onde uma parte típica de dado é escrita ao lado do fluxo, devendo ser precisa e a direção da seta indica o sentido em que os dados estão tomando.


➤ Processo que transforma a estrutura de dados ou a informação contida nos dados. Seu nome deve ser uma instrução significativa com um número de referência.

➤ Depósito de dados é o depositório de dados temporal, sendo um outro nome para arquivo.

➤ A entidade de origem/destino mostra de onde o dado requerido pelo sistema vem e para onde vai.



O DFD para um sistema de porte razoável pode consistir em dezenas de bolhas. Assim, uma solução seria particioná-lo de maneira *top-down* ou *bottom-up*, sem comprometer sua integridade; o que facilitará muito o trabalho do analista.

Figura  Diagrama de Fluxo de Dados.

2.2.1.1. Nivelando um Diagrama de Fluxo de Dados

Uma partição razoável para se escolher é aquela que reflete a maneira como o usuário vê seu trabalho tão bem quanto o analista vê sua análise.

Qual o grau de nivelamento que um DFD deverá ter? O propósito do nivelamento é particionar um sistema grande sem comprometer sua integridade. Cada atividade do DFD se decompõe em atividades de nível mais baixo até que nenhuma decomposição mais seja possível, ou seja, particionar o sistema grande em unidades gerenciáveis .

A regra que comanda quantas “bolhas- filhas ” uma “bolha- mãe” deve ter é a legibilidade, evitando, assim, que o digrama se torne difuso. Mas, se houver somente duas gerações por bolha, não atingirá o nível mais baixo.

Existem alguns sintomas de erros comuns num DFD como bolhas que criam magicamente saídas de dados que não possuem ; arquivos apenas de saída e fluxo de dados sem destino final, os quais entram no sistema, mas nunca são usados.

DFD mal particionados podem ser: fluxo de dados e procedimentos sem nomes e *interfaces* excessivamente complicadas.

2.2.1.2.Os Erros Comuns

É importante evitar erros comuns, como:

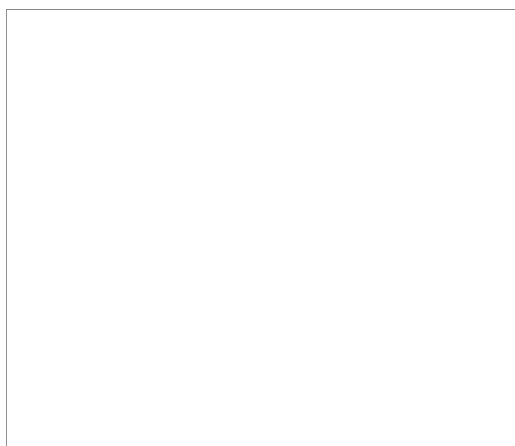
- Fluxos cruzados de dados não cruzando linhas de fluxos de dados . É claro que à medida que o refinamento ocorre é provável que o refinamento horizontal/vertical também ocorra, reduzindo a complexidade do DFD.
- Buracos negros como os processos com entrada e sem saída e depósitos de dados somente para gravação.
- Geração espontânea como os processos com saídas e sem entradas e depósitos de dados somente para leitura.
- Muitos fluxos de dados podendo significar que o diagrama é forte em acoplamento e fraco em coesão.

2.2.1.3.Estratégias de Refinamento do DFD

Um modelo nivelado, apresentado em pedaços é entendível, apresentável e passível de manutenção. Num modelo não nivelado, as informações certamente se tornarão incompreensíveis. Assim, a meta do refinamento é produzir uma especificação do sistema clara, precisa e completa .

Em suma a estratégia do refinamento é criar uma seção de documentação enquanto processos não primitivos ainda existirem num DFD.

2.2.2. Diagrama Entidade Relacionamento



É a notação fundamental para modelagem de dados, e constitui num conjunto de componentes primários para o diagrama E-R , que são os objetos de dados, os atributos, as relações e os vários identificadores de tipos. Seu propósito principal é representar os objetos de dados e seus relacionamentos, onde os objetos de dados são representados por retângulos rotulados e suas relações por diamantes losangos. As conexões entre objetos de dados e seus relacionamentos são estabelecidos usando-se uma série de linhas de ligação especiais.

Figura 2.2 – Diagrama Entidade Relacionamento

Além da notação básica, o analista pode representar hierarquias de dados- objetos- tipo. Elas são análogas à notação estrutural dos objetos OOA. A notação E-R constitui um mecanismo que representa a associatividade entre os objetos.

A modelagem de dados e o diagrama de entidade- relacionamento oferecem ao analista uma notação concisa para examinar dados dentro do contexto de uma aplicação de processamento de dados.

Os objetos são modelados e definidos de acordo com sua função e não em sua forma física. Assim, quando os atributos de dois objetos são similares , os objetos desempenham a mesma função e, de modo inverso, quando os atributos são diferentes, um novo objeto é requerido. Cada instância de um objeto é descrita pelo conjunto idêntico de elementos de dados, variando seus valores.

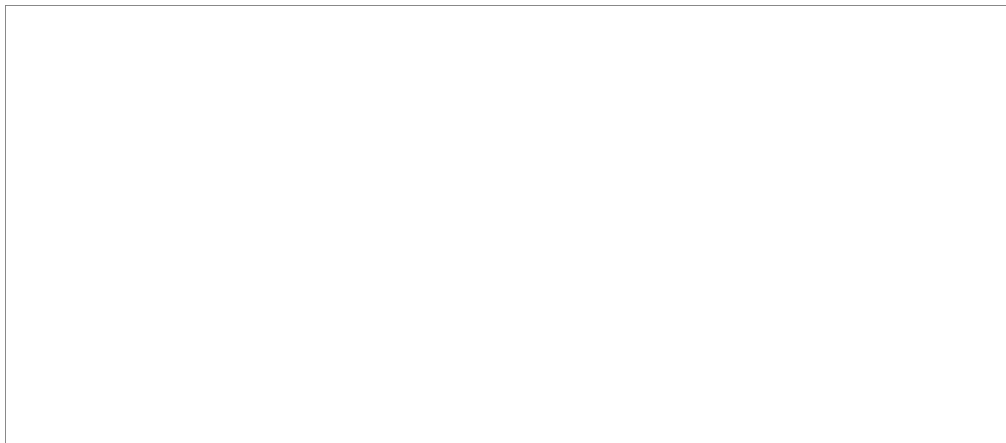
Os relacionamentos conectam objetos e ajudam o analista a decidir quais elementos de dados pertencem a quais objetos. Um relacionamento pode conectar um, dois ou vários objetos, mas a maioria dos relacionamentos conecta no mínimo dois objetos; sendo possível também, que um objeto se relacione consigo mesmo.

Múltiplos relacionamentos podem existir entre os mesmos objetos, como por exemplo num sistema de entrada de pedidos onde “colocar” e “consultar” são relacionamentos que poderiam conectar os objetos “pedido” e “cliente”. Mas é importante ter cuidado com relacionamentos múltiplos que contribuam para o aumento da complexidade do modelo.

2.2.2.1. Refinamento do DER

Num sistema simples não é necessário muito refinamento. Porém, num sistema complexo, a especificação mais detalhada é necessária, com uma visão geral e indicadores para localizar detalhes específicos , ou seja, um modelo nivelado.

A meta do refinamento é produzir uma especificação do sistema clara e completa, verificando se. Assim, somente os objetos e relacionamentos utilizados pelos processos devem ser mostrados.



É importante determinar o melhor tipo de relacionamento que as entidades deverão ter, contribuindo para a redução da complexidade do modelo. E assim os modelos mostrados abaixo devem ser evitados .

Figura 2.3 –Exemplo de DER complexo diversidade de relacionamentos entre entidades.





Figura 2.4- Exemplo de DER complexo com muitas regras entre entidades

2.3. Projeto de Software

O projeto é a primeira das três fases de desenvolvimento do *software* - projeto, codificação e teste e sua importância está na construção de *software* com qualidade; pois serve de base para os passos de engenharia e manutenção do mesmo. “É o processo de se aplicar várias técnicas e princípios para se definir um processo ou sistema com detalhes suficientes que permitam sua realização física, sendo aplicado independente do paradigma.” [TAYLOR59]. Pode se dividir em:

- Projeto de Arquitetura que é a descrição de alto nível onde a aplicação é dividida em subsistemas e a descrição é feita em forma de comunicação e concorrência. Seu objetivo é desenvolver uma estrutura de programa modular e representar os relacionamentos de controle entre seus módulos.
- Projeto Detalhado onde são detalhados os componentes de cada subsistema, sendo que este refinamento envolve a escolha de mecanismos globais para a tomada de decisões ; ajustes das estruturas de classes; descrição de métodos e algoritmos ; empacotamento de classes e associação em módulos.
- Projeto Modular Efetivo que reduz a complexidade, facilita a mudança e a implementação, na medida que estimula o desenvolvimento de diversas partes do sistema em paralelo. Uma produto da modularidade é a independência funcional que facilita o desenvolvimento dos módulos , porque a função pode ser dividida e as *interfaces* simplificadas. É medida usando coesão e acoplamento, onde a coesão é a medida da força funcional relativa de um módulo e o acoplamento é a medida da independência entre os módulos.
- Projeto de Dados é o primeiro dentre as três atividades de projeto realizadas durante a Engenharia de *Software*, tendo uma profunda influência na qualidade do *software*. Sua atividade primordial é a seleção de estruturas de dados que são identificadas nas fase de especificação e requisições.
- Projeto Procedimental que ocorre depois que a estrutura de dados e de programa foram estabelecidos, devendo especificar os detalhes procedimentais de maneira clara e evidente.

Algumas diretrizes para o projeto e qualidade do *software* são:

- Organização hierárquica de controle entre os componentes do *software*;
- Projeto deve ser modular e estes módulos devem apresentar características funcionais independentes;
- *Interfaces* que reduzam a complexidade de conexões entre módulos e o ambiente externo.

As diretrizes acima, juntamente com os aspectos de abstração, modularidade, refinamento ,arquitetura de *software*, hierarquia de controle, estrutura de dados, procedimento de software e ocultação de

informações certamente farão a diferença entre fazer com que o programa funcione e fazê-lo certo, como também lançarão os projetos modulares.

2.4. Documentação do Projeto

É a descrição de projeto completa do *software*, onde as seções são concluídas à medida que o projetista refina sua representação do *software*.

2.5. Componentes de Programa e Interface

Um aspecto importante na qualidade de *software* é a modularização, ou seja, a especificação de componentes de programa (módulos) que são combinados para formar um programa completo.

Embora um componente de programa seja uma abstração de projeto, ele deve ser representado no contexto da linguagem de programação com a qual o projeto será implementado; onde o módulo de nível mais elevado a partir do qual todo o processamento se origina e todas as estruturas de dados se desenvolvem deve ser o primeiro componente a ser identificado. E assim que os componentes de programa tiverem sido especificados, o projeto em desenvolvimento poderá ser examinado e avaliado.

2.5.1. Modularização do Projeto

A modularidade contribui muito para a manutenibilidade, sendo, então, um fator importante na prevenção de manutenção corretiva e aperfeiçoamento do código. Sendo que sua meta é fazer cada rotina como uma caixa preta que tem uma *interface* simples e uma funcionalidade bem definida. Porém, o objetivo da perfeita modularidade é difícil de se realizar com rotinas individuais, pois elas não especificam perfeitamente o porque particionam os dados com outras rotinas.

2.5.2. Coesão

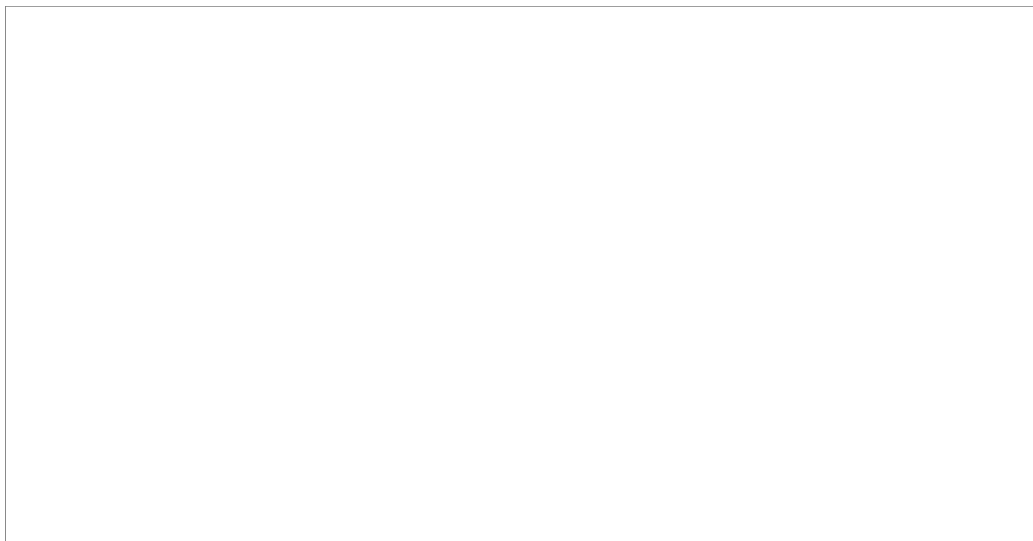
É o grau para o qual os elementos dentro de um módulo contribuem para realizar um propósito único. Pode ser do tipo:

- **Coesão Funcional:** onde um módulo contém elementos que contribuem para a execução de uma e apenas uma tarefa relacionada ao problema. Um exemplo seria um módulo executando uma função que pode ser resumido por um nome verbo-objeto, como: LEIA registro de cliente.
- **Coesão Sequencial** onde os elementos do módulo estão envolvidos em atividades tais que os dados de saída de uma atividade servem como dados de entrada para a próxima. Um número de funções de linha de montagem mostram esta coesão como: EFETUAR TRANSAÇÃO E [USÁ-LA] PARA ATUALIZAR O REGISTRO.
- **Coesão Comunicacional** onde os elementos do módulo contribuem para atividades que usam a mesma entrada ou saída. Um número de funções não-sequenciais trabalhando nos mesmos dados, como no exemplo: CALCULAR VALOR MÍNIMO E MÁXIMO DE MENSALIDADES.
- **Coesão Procedural** onde os elementos do módulo estão envolvidos em atividades diferentes e possivelmente não relacionadas, nas quais o controle flui de uma atividade para outra. Deve-se procurar por nomes de procedimentos ou fluxos, como: CADASTRO DE CLIENTES.

- **Coesão Temporal** onde o módulo envolve atividades de tempo, composto de funções parciais cuja única relação é que todas as atividades acontecem a uma certa hora. São nomes relacionados com o tempo como, por exemplo, FIM DE JOB.
- **Coesão Lógica** onde os elementos contribuem para atividades da mesma categoria geral e as atividades a serem executadas são selecionadas fora do módulo. Nome com um propósito geral, como VERIFICAR DADOS DO CLIENTE, sendo necessário um flag para usar este módulo.
- **Coesão Coincidental** onde o módulo contribui para atividades sem relação significativa entre si, ou seja, suas atividades não estão relacionadas por fluxos de dados ou de controle. Um exemplo seria ROTINA DE PROCESSAMENTO, ou seja, usa um nome pouco significativo, sendo necessário também um *flag*.

2.5.2.1.Comparação entre níveis de Coesão

A coesão é a medida da força do relacionamento dos elementos de um módulo, agindo como uma corrente, ou seja, mantendo juntas as atividades de um módulo. Assim, se todas as atividades de um módulo estão relacionadas em mais de um nível de coesão, o módulo tem a força do nível mais forte de coesão e se as atividades internas de um módulo são mantidas juntas por diferentes níveis de coesão, o módulo tem o poder somente do mais fraco.



A tabela abaixo apresenta um resumo das qualidades específicas de cada tipo de coesão.

Tabela 2.1- Comparação entre os níveis de coesão [PAGE80].

O nível de coesão de um módulo depende das atividades que executa e leva para seu superior e é independente de onde o módulo aparece no diagrama de estrutura.

2.5.2.2.Coesões Aceitáveis

A idéia de coesão foi introduzida por Wayne Stevens, Glenford Myers e Larry Constantine em 1974 [PAGE80]. E algumas idéias são desenvolvidas desde então. Alguns níveis de coesão considerados aceitáveis são :

- **Coesão funcional** é o melhor e mais forte tipo de coesão e ocorre quando uma rotina realiza uma e somente uma operação e quando uma rotina contém operações

que podem ser realizadas em uma ordem específica e não executam a função completa quando trabalham juntas.

➤ A coesão de comunicação ocorre quando operações em uma rotina fazem uso em algum dado e não são relatados em qualquer outro jeito.

➤ A coesão temporal ocorre quando operações são combinadas dentro de uma rotina, porque todas elas estão sendo executadas em algum tempo.

2.5.2.3.Coesões Não Aceitáveis

Alguns tipos de coesão não são aceitáveis, pois elas resultam em códigos pobremente organizados, difíceis para depurar e modificar. Alguns tipos são :

➤ Coesão Procedural que ocorre quando operações na rotina não são executadas na ordem específica.

➤ Coesão Lógica ocorrendo quando muitas operações são suspeitas dentro de alguma rotina e uma das operações são selecionadas pelo *flag* de controle passado.

➤ Coesão Coincidente: ocorre quando as operações na rotina tem uma relação não observadora mútua. Outros nomes são Coesão Caótica ou Não Coesão.

2.5.3. Acoplamento

Método para medir a qualidade de um projeto , ou seja, o grau de interdependência entre os módulos. Assim, é importante minimizar o acoplamento, indicando um sistema bem particionado.

Um baixo acoplamento entre os módulos pode ser obtido eliminando relações desnecessárias , reduzindo o número de relações necessárias e enfraquecendo a dependência das relações necessárias. Existem cinco tipos de acoplamento que são:

➤ Acoplamento de dados onde os módulos se comunicam por parâmetros . É a comunicação de dados necessária entre os módulos.

➤ Acoplamento de Imagem onde os módulos são ligados por imagem se eles se referem à mesma estrutura de dados. Tende a expor o módulo a mais dados do que realmente necessita, com possíveis conseqüências ruins.

➤ Acoplamento de Controle onde um módulo passa para o outro um grupo de dados que controle a lógica interna do outro. O módulo subordinado não é uma caixa preta .

O Acoplamento de Controle às vezes está disfarçado em uma forma denominada Acoplamento Híbrido, que pode causar problemas desastrosos na manutenção, pois resulta na indicação de vários significados para várias partes do domínio de um grupo de dados.

➤ Acoplamento Comum onde dois módulos se referem à mesma área de dados. Não é aconselhável pois o excesso de uso de dados comuns degrada a idéia de modularidade ao deixar os dados abandonarem os limites escritos de um módulo.

➤ Acoplamento de Conteúdo (ou Patológico) onde um módulo faz referencia ao interior do outro. Assim, um módulo sabe o conteúdo e implementação do outro, assim, a maioria das linguagens de alto nível não permitem este tipo de acoplamento.

Dois módulos podem estar acoplados em mais de uma maneira, sendo então, definidos pelo pior tipo de acoplamento que representem. Por exemplo, se dois módulos são ligados por acoplamento de imagem e comum, a característica deles será o acoplamento comum.

Alguns exemplos de descrições boas e ruins de acoplamento: acoplamento de dados simples (chamado de acoplamento normal) ; acoplamento de estruturas de dados; acoplamento de controle; acoplamento de dados globais (chamado de acoplamento comum ou global) e acoplamento patológico.

O acoplamento patológico (ou de Conteúdo) é inaceitável, pois falha nos critérios de tamanho, visibilidade e flexibilidade.

2.5.4. Perda de Acoplamento

O degrau do acoplamento se refere ao poder de conexão entre duas rotinas. Acoplamento é o complemento de coesão. Coesão descreve como os conteúdos internos de uma rotina são relatados mutuamente. Acoplamento descreve como a rotina está relatada dentro de outra rotina. Assim, bons acoplamentos entre rotinas podem ser perdidos de uma rotina que pode facilmente ser chamada por outras rotinas.

Existem alguns critérios utilizados para usar acoplamentos entre rotinas, que são: número, direção e distinção de conexões entre rotinas e como mudar a conexões entre estas rotinas.

2.5.5. Aumentar a Qualidade

Os processos que elaboram os produtos de alta qualidade logo no início do desenvolvimento- especialmente a análise e projeto - podem reduzir a quantidade de erros e melhorar a qualidade, através da facilidade de utilização, portabilidade, manutenabilidade .

2.5.6. Elevar a Manutenabilidade

Várias forças como clientes, competição, reguladores, demonstradores e tecnólogos afetam esse conjunto de requisitos permanentemente em mudança.

Funções serão necessárias para se criar um objeto, conectar objetos a outros, calcular um resultado e proporcionar a monitoração das atividades, sendo que o grau de sofisticação das funções é instável e sujeito às restrições: capacidade, planejamento, orçamento e pessoas.

2.5.7. Reutilização

A reutilização do *software* é raramente praticada com eficiência, pois seus benefícios requerem um investimento capital para: criar os componentes reutilizáveis em primeiro lugar; realizar níveis mais altos de garantia da qualidade, do que os esperados para componentes de *software* de uso único e manter bibliotecas, paginadores e outras facilidades. Estes benefícios são a qualidade e a produtividade.

As razões porque as pessoas não estão usando a reutilização são: os livros de engenharia não promovem ou discutem sobre reutilização e as organizações não oferecem incentivos à utilização da mesma, além do desafio de solucionar um problema de uma única forma, inibindo, assim, a reutilização de um componente alternativo.

2.5.7.1.Níveis de Reutilização

Reutilizar o código pode tomar diversas formas, como por exemplo, fazer uma chamada a uma subrotina para um módulo numa biblioteca; cortar e colar o código fonte, que é a forma mais primitiva de reutilização; incluir ao nível fonte (incorporar código fonte de uma biblioteca num programa); herança (única ou múltipla) para proporcionar uma base técnica para reutilização; encadeamentos binários e chamadas em tempo de execução.

Um modelo de projeto existente pode ser reutilizado como ponto de partida para um projeto diferente, o que seria apropriado se um sistema fosse movido de um ambiente *batch* para ambiente *on line*. Assim, o gerenciamento é um dos ingredientes mais importantes para aquisição de altos níveis de reutilização.

2.6. Complexidade de Código

O *software* é medido para indicar a qualidade do produto; avaliar a produtividade dos desenvolvedores e benefícios derivados dos novos métodos e ferramentas de software. Tudo isso para evitar falhas que podem ocorrer tanto no começo quanto na fase do projeto e desenvolvimento do *software*.

As métricas ajudam no entendimento do processo técnico de desenvolvimento de um produto, a fim de melhorar sua qualidade e fornecer a compreensão sobre o processo de Engenharia de *Software* e o produto a ser desenvolvido. Podem ser divididas em categorias, descritas a seguir:

2.6.1. Métricas de Qualidade do Software

Incluem a complexidade e tamanho do programa e a modularidade efetiva, sendo descritas como a garantia estatística da qualidade, pois avaliam o *software* através da manutenabilidade, integridade e usabilidade do produto e representam medidas indiretas, pois não medem propriamente a qualidade e sim, a sua manifestação. O fator complicador é a relação exata entre a variável que é medida e a qualidade do *software*.

A seguir são descritas um conjunto de métricas de *software* que podem ser aplicadas na avaliação quantitativa da qualidade de *software*, que são:

2.6.1.1.Métrica de Halstead

Esta métrica provavelmente é a mais conhecida e estudada das medidas de complexidade de *software*. A seguir, tem-se alguns pontos principais para esta métrica:

- Propõe as primeiras leis analíticas do *software*;
- Indica as leis quantitativas ao desenvolvimento do *software* ;
- Usa um conjunto de medidas primitivas, que pode ser derivado depois que o código é gerado, para desenvolver expressões para o comprimento global do programa, o volume mínimo potencial para um algoritmo e o real, o nível do programa e da linguagem;
- Propõe que cada linguagem possa ser categorizada por um nível de linguagem um que irá variar conforme a mesma;
- Formulou a teoria de que o nível de linguagem é constante para determinado trabalho. Um nível de linguagem implica em um nível de abstração na especificação do procedimento; então, a linguagem de alto nível permite a especificação de um código em um nível de abstração mais elevado.

➤ É lingüística e baseada em dois quantificadores N1 e N2 que caracterizam qualquer programa e que podem ser determinados antes de qualquer código escrito. E quando o código existe, pode-se calcular o tamanho atual de Halstead (N), através da fórmula:

$N = N1 + N2$, que é uma medida da complexidade do programa .

Halstead demonstra que o comprimento de N pode ser calculado de :

$N = n1 \log_2 n1 + n2 \log_2 n2$, onde :

n1 - número de operadores distintos no programa;

n2 - número de operandos distintos que aparecem no programa;

N1 - número total de ocorrências de operadores;

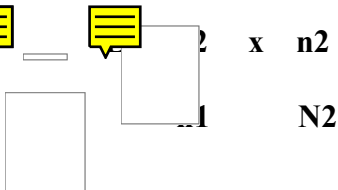
N2 – número total de ocorrências de operandos.

E o volume do programa pode ser definido como:

$V = N1 \log_2 (n1 + n2)$, notando que V representa o volume da informação, variando conforme a linguagem de programação.



Relação entre o volume da forma mais compacta de um programa e o volume do programa real é definido pelo índice volumétrico L e este índice será sempre menor que 1 .



2.6.1.2.Complexidade Ciclomática de McCabe

Pode ser definida como:

$M = L - N + 2D$, onde:

L - número de ligações no gráfico.

N - número de nós do gráfico.

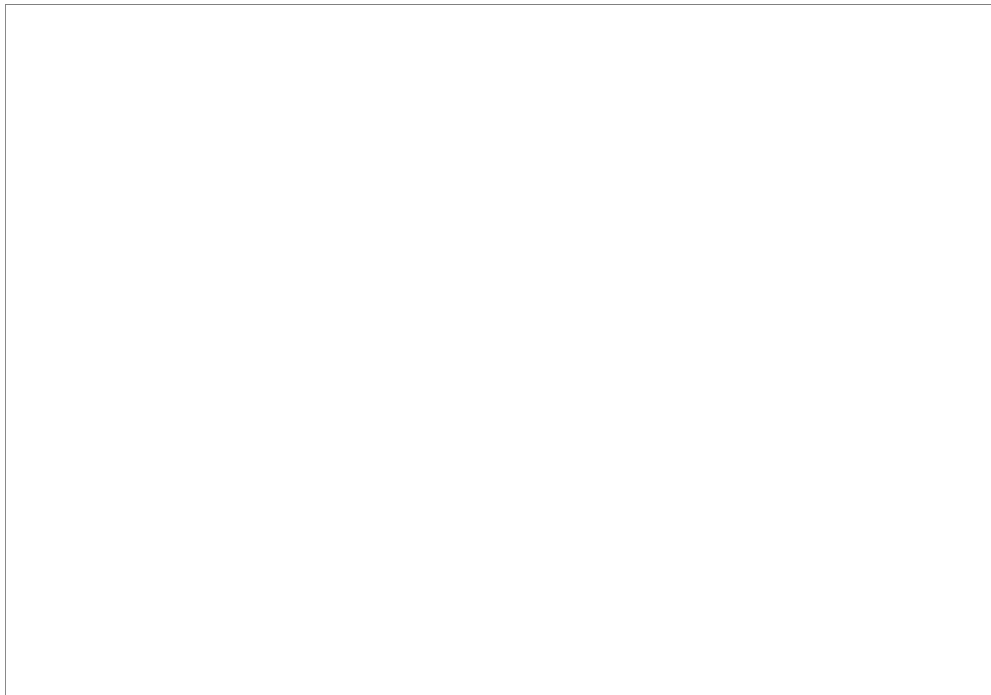
P - número de partes desconectadas no gráfico.

Ou

$M = C + 1$, onde:

C - número de decisões binárias no gráfico.





Esta métrica foi proposta por Thomas McCabe e baseia-se numa representação do fluxo de controle de um programa, definindo uma medida de complexidade que se baseia na Complexidade Ciclométrica de um gráfico de programa para um módulo que pode ser usado para computar a métrica da Complexidade Ciclométrica $V(G)$; consiste em determinar o número de regiões num gráfico planar, onde uma região pode ser informalmente descrita como uma área incluída num plano do gráfico. Enfim, é uma métrica de software que proporciona uma medida quantitativa da complexidade lógica de um programa, ou seja, da dificuldade de fazer testes e uma indicação da confiabilidade final do mesmo. Quando usado no contexto do método de teste do caminho básico, o valor da Complexidade Ciclométrica define o número de caminhos independentes do conjunto básico de um programa e oferece um limite máximo para o número de testes que deve ser realizado para garantir que as instruções sejam executadas pelo menos uma vez.

Figura  Complexidade do gráfico de Controle [PRESSMAN95].

Na figura acima, o gráfico é utilizado para descrever o fluxo de controle, onde cada letra dentro do círculo representa uma tarefa de processamento e o fluxo de controle são as setas de ligação. Assim, por exemplo, a tarefa de processamento a pode ser seguida pela b, c ou d, dependendo das condições testadas. Como o número de regiões é determinado pela soma das áreas delimitadas e a não delimitada, conclui-se que o gráfico possui cinco regiões e assim, tem-se uma Complexidade Ciclométrica $V(G) = 5$.

Estudos indicam relações distintas entre a Métrica de McCabe e o número de erros existentes no código fonte, bem como o tempo exigido para descobrir e corrigir tais erros.

M McCabe defende a afirmação que $V(G)$ pode ser usada para proporcionar uma indicação quantitativa do tamanho máximo modular. $V(G)=10$ parece ser um limite superior prático para o tamanho modular, pois quando a complexidade dos módulos ultrapassa esse número torna-se difícil testar um módulo.

Tem duas bases na teoria dos grafos e é computada em três algoritmos:

Número de regiões do gráfico de fluxo corresponde à Complexidade Ciclométrica;

$V(G)$ é definida como:

$$V(G) = E - N + 2, \text{ onde:}$$

E - número de ramos do grafo do fluxo

N – número de nós do grafo.

$V(G) = P + 1$ onde :

P - número de nós preditivos contidos no grafo .

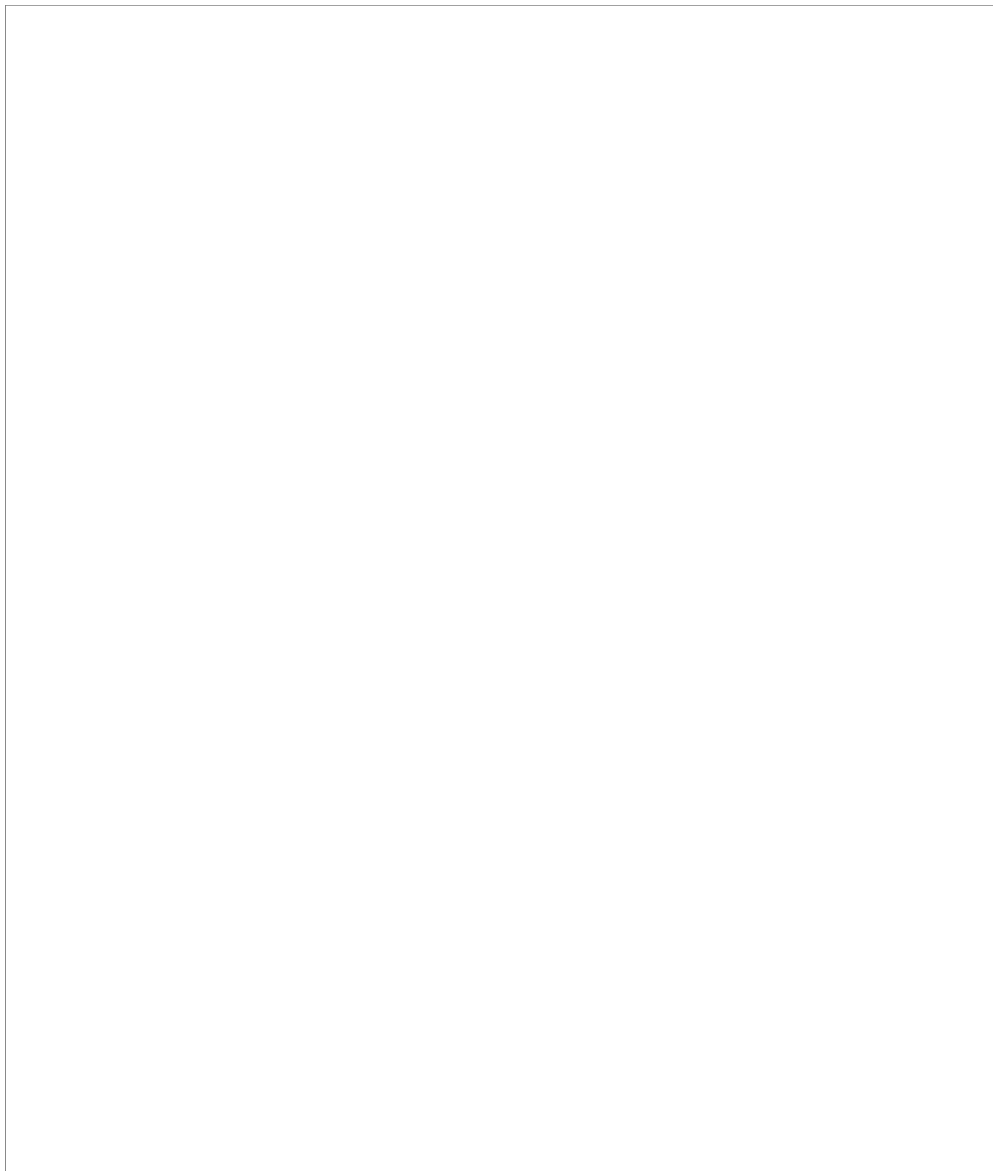
Na figura 2.6 abaixo a Complexidade Ciclomática pode ser obtida usando cada um dos algoritmos, obtendo a seguinte resposta:



Gráfico de fluxo tem 4 regiões ;



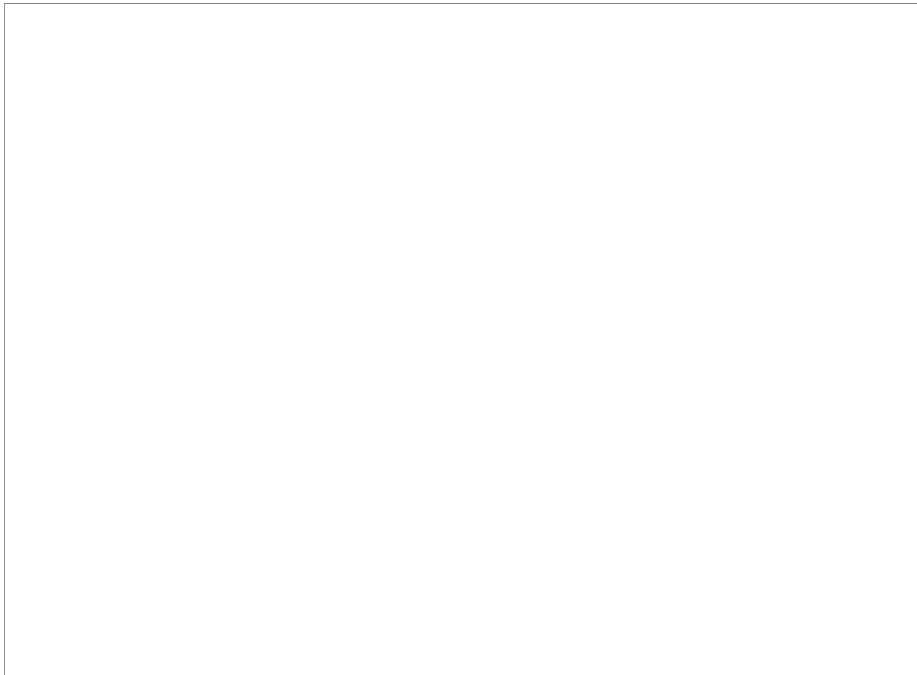
$V(G) = 11 \text{ ramos} - 9 \text{ nós} + 2 = 4;$



$V(G) = 3 \text{ nós preditivos} + 1 = 4.$ O valor de $V(G)$ oferece um limite máximo para o número de caminhos independentes que constitui o conjunto básico, o que implica no limite máximo no número de testes que deve ser projetado e executado a fim de garantir a cobertura de todas as instruções de programa.

Figura 2.6- Complexidade Ciclomática [PRESSMAN95]

2.6.1.2.1.



Exemplo da Métrica McCabe com Código

Figura 2.7- Exemplo em C .

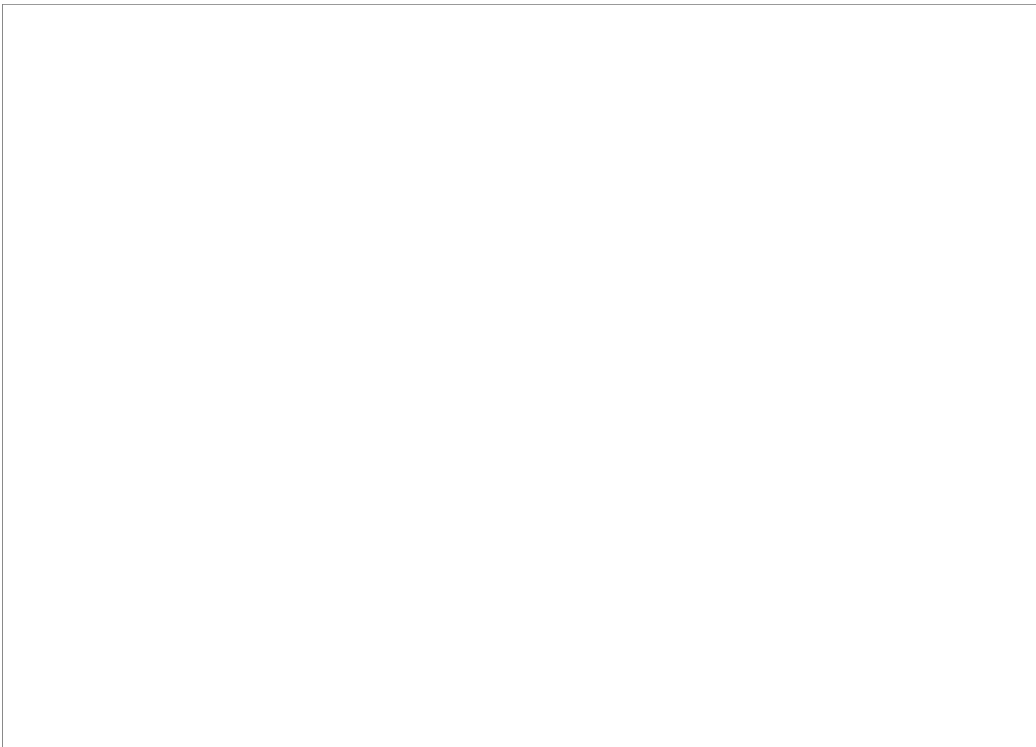


Figura 2.8- Grafo do Fluxo de Controle

2.6.2. Métricas de Produtividade e Qualidade do Software

Referem-se a uma variedade de medidas que proporcionam a qualidade e produtividade do *software*, fornecendo quão estreitamente o mesmo conforma-se às exigências explícitas e implícitas do cliente.

2.6.3. Métricas Orientadas ao Tamanho

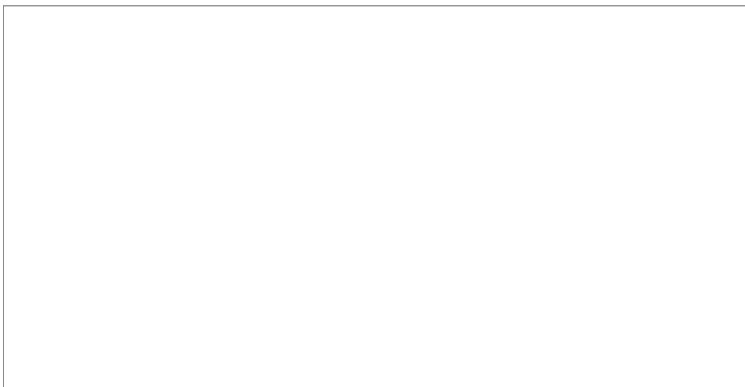
São medidas direto do *software* e do processo por meio do qual ele é desenvolvido, através de dados brutos contidos em uma tabela, possibilitando o desenvolvimento de um conjunto de métricas de qualidade e produtividade orientadas ao tamanho para cada projeto.

2.6.4. Métricas Orientadas à Função

São medidas indiretas do *software* e do processo por meio do qual ele é desenvolvido, concentrando-se na funcionalidade do programa.

Foi primeiro proposta por Albrecht [ALBRECHT79] que sugeriu uma abordagem à medição da produtividade chamada de método ponto- por- função, idealizada para ser usada em aplicações de sistemas de informação.

2.6.5.



Gerenciamento do Software

Figura  Gerenciamento do Software Relacionado com sua Construção [MCCONNELL93].

Como encorajar boas práticas de codificação?

Padrões não podem ser impostos para todos, mas podem evitar o mau uso de alternativas. Algumas técnicas para realizar boas práticas de construção são: determinar duas pessoas para qualquer parte do projeto; rever toda a linha de código a fim de melhorar sua qualidade; requisitar código que anuncie o final do mesmo; fixar um bom exemplo de código para revisão; dar ênfase à listagem de códigos públicos; gratificar o código bom.

O projeto de *software* é dinâmico e por isto, é importante o gerenciamento da configuração, que é uma prática de mudanças sistemáticas em que o sistema pode manter sua integridade durante todo o tempo. Se as mudanças de código não são controladas, a mudança de alguma rotina poderá ser problemática.

Durante o desenvolvimento, podem aparecer idéias sobre como melhorar o projeto do sistema e evitar

problemas futuros , mas é importante o controle do código fonte para comparação de versões, quando um novo erro surgir e também para o *backup*.

Outros caminhos a serem seguidos na construção do *software* são: estabelecer objetivos; formalizar requisições de *software*; avaliar os detalhes de baixo nível; usar diferentes técnicas de avaliação e comparar os resultados e reavaliar as técnicas utilizadas periodicamente, além do uso das métricas.

2.6.6. Construção de Rotinas

Muitos detalhes de baixo nível dentro da construção de rotinas não precisam ser controlados em qualquer ordem particular, mas a maioria das atividades como o projeto e checagem da rotina e o código são executados na ordem da figura abaixo.

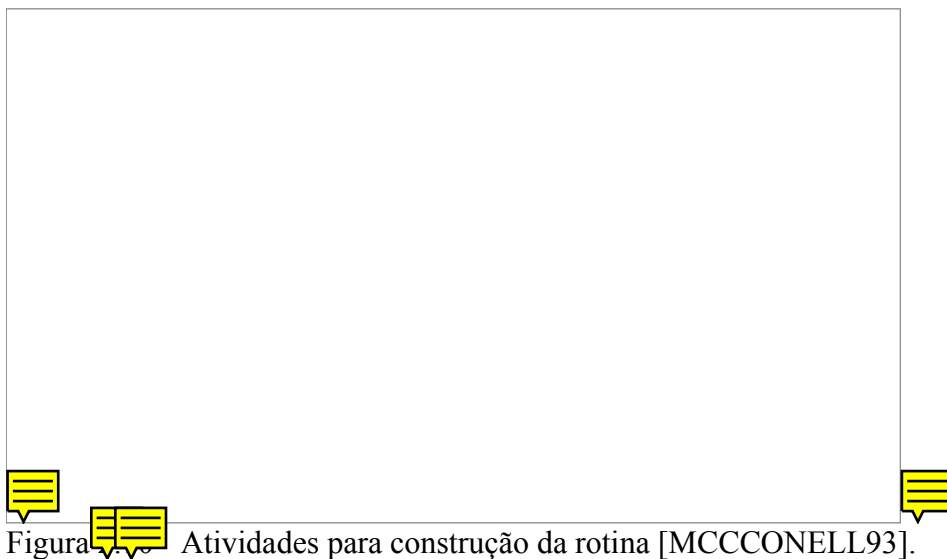


Figura 2.6.6. Atividades para construção da rotina [MCCCONELL93].

O primeiro passo para a construção de uma rotina é o projeto, mas antes de se começar a trabalhar com ela, é bom verificar se a mesma está bem definida, para, em seguida definir o problema a ser resolvido pela rotina. E logo depois , deverá ser testada. E uma vez que o projeto esteja pronto, é preciso implementá-lo. Então é preciso escrever a declaração da rotina, que poderá ser uma função em Pascal, C ou qualquer outra linguagem. Em seguida, é importante checar se a construção está correta, através da compilação e posterior remoção dos erros.

Algumas razões para se criar rotinas são :

- Redução da Complexidade através da minimização do código, fornecendo, assim, a manutenabilidade.
- Evitar Duplicação de Código, o que implicaria em erro na decomposição. E a solução é retirar o código duplicado de ambas as rotinas e colocar uma versão de código dentro da rotina, para, então chamar a parte colocada dentro da nova rotina.

- Limitar os Efeitos de Mudanças de áreas isoladas como as dependências de *hardware*, I / O, estruturas de dados complexas e regras de negócios.
- Ocultar Sequências, ou seja, a ordem em que os eventos acontecem e são processados, através da criação de eventos em que um ou outro seja executado primeiro.
- Fornecer Performance, otimizando o código em uma posição, ao invés de várias, beneficiando todas as rotinas e tornando o algoritmo mais eficiente e rápido.
- Fazer pontos centrais de controle, a fim de manter o controle para cada tarefa em um lugar, como o conhecimento do número de entidades em uma tabela, ou o controle de dispositivos de *hardware* – discos, impressoras, *plotters*. Um exemplo é a criação de uma rotina para ler do arquivo e outra rotina escrever.
- Ocultar Estruturas de Dados fornecendo um valioso nível de abstração que reduz a complexidade do sistema, pois centralizam as operações de estruturas de dados em um ponto, reduzindo as chances de erros.
- Ocultar Dados Globais fornecendo vários benefícios, pois pode-se mudar a estrutura dos dados, sem mudar o programa e monitorar o acesso aos dados.
- Ocultar pontos de operações isolando os pontos em rotinas para se ter certeza que o código está correto. E assim, a mudança do programa não traumatiza a rotina quando a melhor estrutura de dados for encontrada.
- Promover a reusabilidade do código em outras partes do programa .
- Planejar para a família de programas para que a modificação ou criação de rotinas não afete o resto do programa .
- Fazer a seção de código legível colocando a seção de código dentro de uma rotina com um nome bem definido.
- Fornecer a portabilidade evitando o uso de rotinas isoladas sem capacidades portáteis, evitando, assim, posteriores falhas em linguagens não padrão e dependências de *hardware*.
- Isolar operações complexas evitando erros em algoritmos complicados, protocolos de comunicação e operações em dados complexos.
- Isolar o uso de funções de linguagens não padrão, pois estas não podem estar disponíveis em ambientes diferentes.
- Simplificar os testes booleanos a fim de sumarizar o propósito do mesmo.

É sempre bom utilizar nomes claros e que descrevam o que a rotina faz. Algumas diretrizes para nomes de rotinas são verbos fortes acompanhados pelo objeto para o nome da *procedure*; para o nome da função é bom usar a descrição do valor de retorno; evitar verbos com significados fracos; descrever o que a rotina faz; estabelecer convenções para operações comuns e não colocar nomes muito longos nas rotinas.

2.6.6.1.Decomposição Top- Down

É a técnica de decomposição de um programa dentro de rotinas, caracterizada pelo movimento de

statement geral que o programa faz para os *statements* detalhados sobre tarefas específicas. É um processo iterativo, porque geralmente não pára depois do primeiro nível de decomposição e sim, para outros níveis; decompõe o programa de uma maneira e vê o que acontece, e somente então decompõe-se de outra maneira e novamente vê o que acontece. E depois de várias tentativas, chega-se à melhor idéia.

2.6.6.2. Decomposição Bottom- Up

Para trabalhar com alguma coisa mais concreta é necessário utilizar a decomposição *bottom up*, identificando o que o sistema necessita; os baixos níveis capazes seus aspectos comuns.

2.6.6.3. Decomposição Top- Down vs Bottom-Up

A diferença entre as estratégias *top down* e *bottom up* é que uma é decomposição e a outra composição; uma começa do problema geral e o quebra em pedaços e a outra começa com as partes, combinando-as para criar a solução geral. O projeto *top down* pode transferir os detalhes da implementação e o *bottom up* resulta da identificação antecipada da utilização de rotinas.

Uma fraqueza da composição *bottom up* é a dificuldade de ser utilizada exclusivamente, pois é mais fácil pegar uma grande concepção e quebrá-la em pequenas, que pegar pequenas concepções e fazer somente uma grande.

Os projetos *top down* e *bottom up* não são estratégias competitivas, são mutuamente benéficas, já que o projeto é um processo heurístico, o que significa que a solução não é garantida para se trabalhar durante todo tempo não contendo, às vezes, um elemento com erro trivial.

2.6.7. Fundamentos do Layout

O Teorema Fundamental da Formatação é que o bom *layout* mostra a estrutura lógica do programa, pois, provavelmente é a chave para sua estrutura. O esquema de um bom *layout* pode ser:

- Representação exata e consistente da estrutura lógica do código através da indentação e uso de espaços em branco;
- Aperfeiçoamento da Legibilidade facilitando a leitura do código;
- Resistência às Modificações, pois a modificação de uma linha do código não requer a modificação das outras.

Pode-se concluir um bom *layout* com o uso de ferramentas de diferentes maneiras, como por exemplo, o uso de espaço em branco para mostrar a estrutura do programa. Porém, o melhor estilo de *layout* depende da linguagem de programação que esteja utilizando.

O *layout* de alguns elementos do programa é um problema para estética, como estruturas de controle que afetam a compreensão e a prioridade prática do programa. Existem alguns pontos para a formatação destas estruturas como:

- Usar linhas em branco entre os parágrafos para melhorar o código, pois abre espaços para os comentários do programa;
- Formatar blocos simples de *statement* que acompanham a estrutura de controle;

➤ Para condições complicadas, colocar condições separadas em linhas separadas onde cada expressão complicada deverá possuir uma linha ;

➤ Evitar *gotos*, pois tornam o código difícil de formatar e dificultam na verificação do programa;

```
X=1;
```

```
Loop1;
```

```
X++;
```

```
If ( X<100) goto loop1;
```

Exemplo 2.1- Exemplo em C do uso de goto.

2.6.8. Ferramentas de Programação

As ferramentas de programação consistem principalmente de ferramentas gráficas que criam diagramas sendo, às vezes embutidas em ferramentas CASE; geralmente permitem expressar um projeto em notações gráficas comuns, como gráficos hierárquicos, diagramas, DER, sendo que algumas suportam somente uma notação, enquanto outras suportam uma variedade.

As ferramentas para código executável são mais ricas que ferramentas para trabalhar com código fonte; ajudam na criação do código e podem ser *linkers*, que são conexões de um ou mais arquivos que o compilador gera dos arquivos fontes; bibliotecas de códigos construídas para escrever códigos de alta qualidade; geradores de código, úteis para fazer protótipos de códigos, demonstrando os aspectos chaves de *interface* com o usuário; macro processadores e depuradores.

2.6.8.1. Editores

São ferramentas disponíveis para trabalhar com o código fonte, são mais ricas e maduras que as disponíveis para o projeto. Consistem em:

➤ Editores com algumas capacidades como mudanças de *string* para múltiplos arquivos. Por exemplo, se surge a necessidade de determinar um nome melhor para a rotina, isto poderá ser feito em muitos arquivos.

➤ Comparações de arquivos, facilitando a correção de erros ou modificações.

➤ Embelezar o código fonte padronizando seu o estilo de indentação, alinhar as declarações de variáveis e cabeçalhos das rotinas, formatar os comentários, etc.

➤ *Templates* que ajudam na utilização de uma idéia simples de tarefas que freqüentemente precisam ser feitas. São uma maneira fácil de encorajar a consistência de código e os estilos de documentação.

2.6.8.2. Browser

Os browsers são um grupo de ferramentas que ajudam na visualização do código fonte, são úteis para mudanças consistentes, procuram e repõem lado a lado um grupo de arquivos, referências e chamadas a estruturas.

Um tipo específico de *browser* pode encontrar múltiplas ocorrências do arquivo da *string* designada, ou variáveis globais; listar variáveis ou rotinas em todos os lugares em que ela está sendo usada e produzir informações sobre rotinas que se chamam mutuamente.

2.7. Analisando a Qualidade do Código

- Checar a sintaxe e semântica do código suplementa a compilação.
- Construir ferramentas que relatam a complexidade das rotinas.
- Reestruturar o Código Fonte , como por exemplo, converter *gotos* em *case*; tradução de código de uma linguagem para outra.
- Controle de Versão para controlar o código fonte e fazer o controle da dependência de estilo.
- Dicionário de Dados com descrições dos nomes das variáveis, tipos e atributos.

2.8. Documentação de Código

Documentação no projeto de *software* consiste da informação do código de origem, usualmente no formulário de documentos separados ou pastas para desenvolvimento de *unit* (UDF), que são documentos informais contendo notas usadas por desenvolvedores, durante sua construção com objetivo de fornecer a trilha de decisões do projeto que não estão documentadas. Em projetos grandes, muitas das documentações estão no exterior do código fonte.

Os detalhes da documentação do projeto estão em um nível baixo de documentação; pois descrevem o nível do módulo ou nível de decisões da rotina e alternativas consideradas.

2.8.1. Estilo de Programação como Documentação

A documentação interna é encontrada com a listagem do programa, sendo, então, um tipo de documentação detalhada; pois é associada com o código. É muito útil para que o código permaneça correto, quando modificado por alguma necessidade , como pequenas melhorias.

A maior contribuição não está nos comentários, mas no estilo de programação com programa estruturado, bons nomes de variáveis e rotinas , *layout* claros e minimização da complexidade de estruturas de dados.

2.8.2. Comentar ou Não Comentar

Comentários são fáceis de serem escritos e ajudam bastante , principalmente na manutenção do *software*.

```
/*-----*/ matriz.c
```

Autor(a): Lilian

Data: 30/04/2000

Este programa calcula a soma de duas matrizes, gerando a matriz resultado.

*****/

Exemplo 2.2- Comentário de um Programa em C.

Para o comentário de linha individuais é importante evitar comentários benignos a si próprio, linhas limites em um única linha e para múltiplas linhas de código. É bom usar a linha limite para anotar as declarações dos dados, notas de manutenção e comentários que marcam o final de blocos.

Muitos programadores concordam que as técnicas de documentação descritas são válidas, mas a evidência científica do valor de qualquer uma delas é forte, quando estas são combinadas.

2.9. Conclusão

O processo de desenvolvimento é dividido em três fases – análise, projeto e implementação. E em todas estas fases podem ser aplicadas técnicas que visem a redução da complexidade e facilitem a manutenibilidade do sistema; principalmente porque grande parte de seu ciclo de vida é consumido com manutenção, aumentando, assim, tempo e custo; tão importantes para as empresas.

É no projeto que as técnicas serão aplicadas para conseqüente construção do sistema e é nele também que a complexidade pode ser reduzida através, por exemplo da modularidade e mecanismos para a tomada de decisões.

Aplicar técnicas e métricas propostas auxiliam na redução da complexidade ainda na fase inicial do processo de desenvolvimento, além de facilitar a vida dos projetistas e desenvolvedores, estimulando, assim, a qualidade do *software*.

3. Complexidade de Modelos Baseados em Objetos

3.1. Introdução

Quando a década de 1980 chegava ao fim, o "paradigma orientado a objeto" da Engenharia de *Software* começava a amadurecer numa abordagem poderosa e prática para o desenvolvimento de *software*, pois a competição entre as empresas estava aumentando devido principalmente à globalização e a necessidade sistemas mais flexíveis e inovadores cresciam proporcionalmente.

A tecnologia orientada a objeto é uma estratégia para organizar sistemas (ferramenta organizacional) como uma coleção de objetos que interagem e combinam dados e comportamentos; sendo independente de linguagens de programação, até o estágio de implementação. É um novo paradigma de desenvolvimento, usando notações gráficas como UML, Booch e OMT para representação dos conceitos e trazendo vantagens como sistemas menores, através do reuso de componentes existentes. É usada para descrever um sistema com diferentes tipos de objetos e as ações que dependem do tipo de objeto que se está manipulando.

O paradigma de orientação a objeto tem se mostrado uma solução promissora para a maioria das aplicações, devido principalmente à reusabilidade do código e, como consequência, pode-se esperar benefícios como o melhor gerenciamento da complexidade, aumento da produtividade e redução de custos a longo prazo.

A programação orientada a objetos, contém objetos que consistem em dados e procedimentos para manipular esses dados; combinam estrutura de dados e funções para criar objetos reutilizáveis e tem como principal vantagem a capacidade de criar módulos que não necessitam de mudanças, quando um novo tipo de objeto é criado, possibilitando ao programador a criação de objetos que herdem características de objetos existentes.

As linguagens mais populares orientadas a objetos são Java, C++ e *Smalltalk*, sendo que esta última é pura linguagem OO, com programas significativamente mais rápidos de se desenvolverem, simplificando o complexo mundo da computação *client server*.

Neste capítulo tem-se uma breve descrição de conceitos da OO, assim como algumas medidas para redução da complexidade do *software* – medidas estas que buscam desenvolver sistemas com maior integridade, manutenabilidade e corretitude.

3.2. Conceitos Básicos sobre a Orientação a Objetos

A análise e o projeto OO são fundamentalmente diferentes do projeto estruturado pois requerem um tipo diferente de pensar sobre decomposição, exteriorizando o domínio mais largamente.

A orientação a objeto enfatiza a importância de identificar com precisão os objetos e suas propriedades, antes de começar a escrever os detalhes das manipulações. Sem essa identificação cuidadosa, é quase impossível ter-se precisão sobre as operações a serem executadas e seus efeitos pretendidos.

A linguagem orientada a objeto possui algumas propriedades como : encapsulamento, ocultação de informação e implementação; retenção de estado, identidade do objeto, mensagens, classes, heranças, polimorfismo, genericidade que serão discutidas posteriormente.

3.2.1. Definição de Objeto

É o conjunto de métodos e variáveis; uma caixa preta que recebe e envia mensagens, contendo código e dados unidos; um componente do mundo real que é mapeado para domínio de *software*; instância individual de uma classe; é definido via classe, que determina tudo sobre ele; estrutura de dados encapsulada que resulta em dados ativos e que pode ser solicitado a fazer coisas ao receber a mensagem. Não pode ser modificado diretamente, o que causaria a adulteração de detalhes sobre como ele trabalha.

O histórico genético de um objeto é definido ao se reconhecer que o objeto deve ser criado, modificado, manipulado, lido de outras formas e possivelmente deletado.

Quando um objeto é representado graficamente em sua concepção de *software*, ele compõe-se de uma estrutura de dados e processos particulares denominados operações. Operações estas que, juntamente com o objeto, proporcionam a modularidade entre os elementos de *software* e que contém construções procedimentais e de controle e que podem ser invocadas por mensagens, que são as formas de comunicação entre objetos e definem a *interface* dos mesmos; sendo que esta é a parte compartilhada do objeto.

As mensagens movem-se através da *interface* do objeto e especificam qual operação é desejada. O objeto recebe a mensagem e determina como as operações solicitadas devem ser implementadas.

Quando um objeto é definido com uma parte reservada e mensagens são invocadas para seu

processamento adequado realiza-se então, a ocultação de informações , ou seja, os detalhes da implementação são escondidos de todos os elementos de programa, que se encontram fora deste objeto.

Um objeto fornece uma encapsulação, por meio da qual uma estrutura de dados e um grupo de procedimentos para acessá-la podem ser postos em serviço, de tal forma que os usuários desse recurso possam acessá-la através de um conjunto de *interfaces* cuidadosamente documentadas, controladas e padronizadas.

3.2.2. Classes, Instância e Herança

As concepções de *software* de objetos do mundo real são divididas em categorias, de forma muito parecida: todos os objetos são membros de uma classe mais ampla e herdam a estrutura de dados e operações particulares, que foram definidas para essa classe. Assim, a classe é definida como um conjunto de objetos, em que cada um tem as mesmas características; é uma matriz da qual os objetos são criados; uma instância de classe mais ampla; é o que projeta e programa o objeto e é o que se cria durante o processamento.

Todos os objetos criados de uma mesma classe tem a mesma estrutura e comportamento e seus métodos e variáveis são idênticos, mas cada objeto tem sua própria cópia do conjunto de métodos e variáveis, sendo que um único conjunto pode ser compartilhado por todos os objetos.

3.2.3. Identificação de Objetos

Os objetos são determinados sublinhando-se cada nome ou cláusula nominal e colocando-os em uma tabela. Se o objeto for requisitado para implementar uma solução, ele faz parte do espaço solução; de outro modo, se o objeto for necessário somente para descrever uma solução, ele faz parte do espaço problema.

Objetos podem ser:

- Entidades externas: que produzem ou consomem informações a serem usadas por um sistema baseado em computador.
- Coisas que fazem parte do domínio da informação do problema.
- Ocorrências ou eventos que ocorrem dentro do contexto de operação do sistema.
- Papéis desempenhados por pessoas que interagem com o sistema.
- Unidades organizacionais que são pertinentes a uma organização.
- Lugares que estabelecem o contexto do problema e a função global do sistema.
- Estruturas que definem uma classe de objetos ou, ao extremo, classes relacionadas de objetos.

Existem algumas características de seleção que devem ser usadas quando o analista examinar cada objeto em potencial, para inclusão no modelo de análise que são:

- Informação retida. O objeto em potencial será útil durante a análise somente se a informação sobre ele precisar ser lembrada, de forma que o sistema possa funcionar.
- Operações identificáveis que possam mudar o valor dos atributos dos objetos de alguma maneira.
- Múltiplos atributos. Durante a análise de requisitos, o foco deve recair sobre informações "importantes". Um objeto com um único atributo pode, de fato, ser útil durante a fase de projeto, mas provavelmente ele será melhor representado com um atributo de um outro objeto, durante a atividade de análise.
- Atributos comuns. Um conjunto de atributos que se aplicam a todas as ocorrências do objeto pode ser definido para o objeto em potencial.
- Operações comuns. Um conjunto de operações pode ser definida para o objeto em potencial e estas operações podem ser aplicadas em todas as ocorrências do mesmo.
- Requisitos essenciais. Entidades externas que aparecem no espaço problema e produzem ou consomem informações essenciais à operação de qualquer solução para o sistema, sendo definidas como objetos, no modelo de requisitos.

Uma descrição de implementação de um objeto oferece os detalhes internos exigidos, mas que não são necessários para sua invocação; ou seja, o projetista deve oferecer uma descrição de implementação e criar os detalhes internos do objeto. A descrição de implementação é composta das informações:

- Especificação do nome do objeto e referência a uma classe;
- Uma especificação da estrutura de dados reservada, com uma indicação dos itens e tipos de dados;
- Uma descrição procedimental de cada operação.

Os atributos descrevem e definem os objetos selecionados para inclusão, no modelo de análise. Assim, para desenvolver um conjunto de atributos significativos para um objeto, o analista poderá estudar uma vez a narrativa de processamento do problema e selecionar aqueles aspectos que razoavelmente "pertencem" ao objeto.

3.2.4. Definição de Operações

Operações podem mudar objetos alterando um ou mais valores de seus atributos; mas devendo ter sempre o "conhecimento" da natureza dos atributos e capacitando a manipulação das estruturas de dados derivadas dos atributos. Possuem algumas categorias, como as operações que manipulam dados de alguma maneira; que realizam uma computação e que monitoram um objeto, quanto à ocorrência de um evento controlador; o que significa que operações adicionais podem ser determinadas, ao se considerar o histórico de um objeto e as mensagens que passaram entre os objetos definidos.

3.2.5. Identidade do Objeto

Cada objeto (independente de sua classe ou estado) pode ser identificado e tratado como entidade de *software* distinta, com algo único, ligado a ele, que o distingue dos outros - o mecanismo identificador do objeto. E este identificador se baseia em nas regras de que o mesmo identificador permanece com o objeto por toda sua existência e dois objetos não podem ter o mesmo identificador.

3.2.6. Mensagens

É a maneira pela qual um objeto emissor se dirige ao objeto alvo, solicitando-o a aplicar seus métodos; é o mecanismo de comunicação estabelecido entre os objetos, onde o objeto é requisitado a executar uma operação ao receber a mensagem que diga ao objeto o que fazer. O objeto alvo responde à mensagem escolhendo primeiramente a operação que implementa a mensagem, executando-a e depois devolvendo o controle ao emissor.

Sua estrutura se compõe de um identificador do objeto alvo onde emissor manterá o identificador em uma variável; o nome do objeto alvo que o emissor executará; quaisquer informações suplementares que o objeto alvo necessitará na execução de seu método.

mensagem: (destino, operação, argumentos)

onde "destino " define o objeto a receber a mensagem, "operação " refere-se à operação que receberá a mensagem e "argumentos" fornece as informações que são exigidas para que a operação seja bem-sucedida.

3.2.6.1.Os papéis dos Objetos nas Mensagens

Um objeto pode ser emissor e alvo de uma mensagem apontado por uma variável dentro de outro objeto ou por um argumento passado ou retornado de uma mensagem, pois há objetos apontando para objetos e comunicando-se com outros objetos.

As variáveis são usadas para guardar informações e seu acesso e atualização são através de métodos do objeto. Cada variável relacionada a um e outro objeto poderá somente ser acessado dos métodos do objeto. Estes métodos são, simplesmente, ações que as mensagens carregam, ou seja, o código que é executado quando a mensagem é enviada para um objeto particular; formando, assim, um anel protetor no núcleo das variáveis. Envia mensagens para os objetos apontados pelas variáveis privativas do mesmo. Podem ser públicos ou privados.

3.2.6.2.Tipos de Mensagens que um Objeto pode receber

São três os tipos de mensagens que podem ser orientadas para o passado, presente e futuro. São descritas a seguir:

- Mensagem informativa onde o objeto recebe informações para que ele próprio se atualize (mensagem de atualização); é orientada para o passado.
- Mensagem interrogativa onde o objeto recebe uma mensagem pedindo a ele que revele alguma informação sobre si mesmo; é orientada para o presente.
- Mensagem imperativa onde há o pedido ao objeto de alguma ação sobre si próprio, sobre outro objeto ou sobre o ambiente em que se encontra; é orientada para o futuro.

3.2.7. Abstração

É o princípio de ignorar os aspectos de um assunto não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais; seleção que um analista faz de alguns aspectos, ignorando outros.

Os tipos de abstração são descritos a seguir:

➤ A abstração de procedimentos é uma forma de abstração usada por analistas de requisitos, projetistas e programadores e é caracterizada como uma abstração função/subfunção; é o princípio de que qualquer operação com efeito bem definido poderá ser tratada como entidade única, mesmo que a operação seja realmente conseguida através de alguma seqüência de operações de nível mais baixo. Um exemplo são os diagramas de estruturas.

➤ A abstração de dados consiste na definição de um tipo de dado conforme as operações aplicáveis aos objetos deste tipo; assim, os objetos podem ser modificados e observados através destas operações. Este tipo de abstração é mais poderoso que a abstração de procedimentos e pode ser usado como base para a organização do pensamento e a especificação das responsabilidades de um sistema.

Ao se aplicar uma abstração de dados, um analista define os atributos e os serviços que manipulam exclusivamente estes atributos que podem ser tratados como um todo intrínseco.

O programador pode usar abstração para notar que duas funções tem tarefas comuns e podem ser combinadas em uma simples função. É uma técnica importante em Engenharia de *Software*, sendo relatada com outra técnica conhecida como encapsulamento, que será discutida a seguir. Assim, o programador poderá focar-se em novos objetos, sem se preocupar com a ocultação dos detalhes.

3.2.8. Encapsulamento

É o agrupamento de idéias relacionadas em uma unidade; uma técnica poderosa de programação que reduz a complexidade e previne contra mudanças intencionais ou não de partes do programa; agrupamento de procedimentos em torno de idéias. Já o encapsulamento orientado a objeto é o agrupamento de procedimentos em torno dos dados, revelando informações e escondendo a implementação. Então, pode-se concluir que o encapsulamento pode ser uma poderosa técnica para domesticar a complexidade do sistema.

Algumas vantagens do encapsulamento podem ser descritas como a diminuição de trabalho no desenvolvimento de um novo sistema; o agrupamento de aspectos relacionados; a minimização do fluxo entre as diferentes partes do trabalho e a separação de certos requisitos específicos que outras partes da especificação podem usar.

3.2.9. Retenção de Estado

Reter informações sobre si mesmo por um tempo indefinido, ou seja, um objeto não morre quando termina sua execução, mas fica de prontidão, preparado para entrar em execução.

O encapsulamento orientado o objeto, a ocultação de informação e implementação e retenção de estado constituem o núcleo da orientação a objeto.

3.2.10. Herança

É a maneira pela qual um objeto se distancia das abordagens tradicionais do sistema, pois permite que se construa o *software* de maneira instrumental, ou seja, primeiro criam-se as classes para atenderem aos casos mais diretos; e em seguida, para tratar os casos especiais, adicionam-se classes especializadas, que herdam da primeira classe. E essas novas classes serão habilitadas a usar todos os métodos e variáveis da classe original. É o mecanismo para expressar a similaridade entre classes, simplificando a definição de classes iguais a outras que já foram definidas; representa generalização e especialização, tornando explícitos os atributos e serviços comuns em uma hierarquia de classe.

A herança permite ao analista especificar funções e processamentos em casos específicos e representar

elementos comuns explicitamente; podendo ser aplicada no início das atividades de análise. Significa reusabilidade, pois não precisa começar um novo programa, pode-se simplesmente, reusar um repertório de classes existentes com comportamentos similares aos necessários no novo programa. E assim, a subclasse criada herda todas as mensagens e comportamento da classe original e a classe original é chamada, então, superclasse da nova classe.

Na herança única cada classe tem apenas uma subclasse direta. O que não acontece na herança múltipla que converte árvores de herança com heranças únicas em um retículo de heranças e onde cada classe pode ter um número arbitrário de superclasses diretas; podendo trazer problemas para o projeto como, por exemplo, a possibilidade de uma superclasse herdar métodos conflitantes de seus múltiplos ancestrais.

A herança não é perfeita quando uma instância candidata de uma classe compartilha a maioria dos atributos da classe e exige todas as operações da classe, bem como operações adicionais que sejam relevantes somente ao membro candidato.

3.2.11. Polimorfismo

É a habilidade de redefinir métodos para classes derivadas; dispositivo pelo qual o nome de um único método pode ser definido sobre mais de uma classe e pode assumir diferentes implementações em cada uma dessas classes; propriedade pela qual uma variável pode apontar para objetos de diferentes classes, em instantes diferentes.

A sobrecarga ocorre quando vários métodos definidos na mesma classe tem aquele nome de símbolo. E, portanto, tanto o polimorfismo quanto a sobrecarga requerem que o método específico a ser executado seja escolhido durante o processamento.

3.2.12. Genericidade

É a construção de uma classe de maneira que uma ou mais das classes que ela usa internamente é fornecida somente durante o tempo de execução.

3.2.13. Acoplamento

É o grau de interdependência entre pessoas. Em OOD é a interconectividade entre suas peças. É importante na avaliação de um projeto, ajudando a focalizar a atenção na alteração de uma parte do sistema para que esta tenha o mínimo impacto sobre as outras partes.

Se uma alteração numa classe de um sistema causar alterações tipo pipoca, ou seja, espalhando-se por diversas classes, isto é uma indicação de muito acoplamento. Assim, o grau ou força de acoplamento entre dois componentes é medido pela quantidade e complexidade das informações transmitidas entre os componentes. O acoplamento pode ser do tipo:

➤ Acoplamento de Interação onde o desejável é um baixo acoplamento, mantendo, assim, a complexidade de uma conexão de mensagem a mais baixa possível. Em geral, se uma conexão de mensagem envolve mais de três parâmetros, examine-a para verificar se ela pode ser simplificada.

Os objetos conectados através de mensagens complexas são fortemente acopladas, onde uma alteração em uma delas leva a alterações em outros. E para minimizar a complexidade de uma conexão de mensagem, deve-se simplificar o número de mensagens enviadas e recebidas por cada objeto individual.

➤ Acoplamento de herança. A herança é uma forma de

acoplamento entre uma classe de generalização e uma classe de especialização, onde uma classe é acoplada à sua classe de generalização, em termos dos atributos e funções que ela herda. Assim, o acoplamento de herança elevada é desejável.

Cada classe de especialização seria uma especialização de sua classe de generalização, obtendo-se, então, um auto-acoplamento. E se uma classe rejeita explicitamente vários atributos da sua classe de generalização, ela não é fortemente acoplada à sua classe de generalização.

3.2.14. Coesão Orientada a Objeto

O critério para o módulo coesão é simples como o critério para a coesão de uma rotina individual, sendo baseado em pacotes de dados e serviços e está no nível de módulo como um todo. A coesão pode ser do tipo:

- Coesão de Serviço onde um serviço realiza uma e somente uma função ou gerencia apenas parte de uma função, já que a realização de múltiplas funções é indesejável. Uma maneira de padronizar o grau de coesão é solicitar ao projetista que dê um nome ao serviço que descreva sua responsabilidade.
- Coesão de Classe onde os Atributos e serviços seriam mais altamente coesivos, com nenhum atributo ou serviço extra e com todos descritivos das responsabilidades de um objeto da classe.

3.3. Análise Orientada a Objeto

A análise orientada a objeto está fazendo um lento, mas firme progresso como método de análise de requisitos, por seus próprios méritos e como complemento a outros métodos de análise; pois ao invés de examinar um problema usando o clássico modelo de entrada- processamento – saída (fluxo de informação) ou um modelo derivado exclusivamente de fluxo de informações hierárquicas, ela introduz uma série de novos conceitos. Assim, seu uso poderá levar à prototipação extremamente efetiva e a técnicas de "Engenharia de *Software* revolucionárias".

O modelo de análise permite escrever uma especificação para o sistema, que será trabalhada no projeto; e, por isso, procura uma forma de comunicação bastante precisa, que servirá para a comunicação entre analista e projetista. Assim, durante a análise deve-se:

- Identificar os objetos nos personagens e nos elementos do sistema, evitando ambigüidades e complexidades desnecessárias;
- Listar os atributos dos objetos com eles formar uma estrutura de dados sólida para descrever o problema;
- Classificar os objetos, obtendo vantagens de simplificação com uso da herança;
- Utilizar estados dos objetos, para descrever a dinâmica e o controle do processo;
- Identificar as operações sobre os objetos, descrevendo-as, se necessário, por diagramas de processo;
- Manter um nível alto de abstração, evitando a contaminação do modelo por elementos de implementação.

Os objetos que são especificados e por fim implementados num projeto em andamento, podem ser catalogados numa biblioteca. Mas como os objetos são reusáveis, com o passar do tempo a biblioteca de objetos reusáveis crescerá e daí quando a OOA for aplicada a novos projetos, o analista poderá trabalhar para especificar o sistema usando os objetos existentes contidos na biblioteca de objetos.

Ao usar objetos existentes durante a análise, o tempo de especificação é reduzido substancialmente e um protótipo rápido do sistema especificado pode ser criado e revisado pelo cliente.

3.4. Modelagem Orientada a Objeto

É a técnica de análise usada para aplicações com dados intensivos, concentrando-se somente nos dados, representando, assim, uma "rede de dados" de um determinado sistema. É útil para aplicações em que os dados e as relações que regerem esses dados são complexas; considera os dados independentemente dos processos que os transformam.

A terminologia da modelagem de dados e parte de sua notação gráfica são idênticas às utilizadas para OOA, porém suas abordagens são diferentes, com um ponto de vista também bastante diferente; pois ambas usam o termo objeto, mas a sua definição é muito mais limitada num contexto de modelagem de dados; ambas descrevem relações entre objetos, mas a modelagem de dados não se preocupa com a forma como essas relações são conseguidas.

A modelagem de dados modela dados, sem se preocupar com os processos que devem ser aplicados para transformar dados. É uma técnica complementar que atende a uma função de análise específica e deve ser combinada com outra abordagem de modelagem que leve em consideração as questões de processamento, a fim de formar um método de análise de requisitos completo. É usada extensivamente em aplicações de banco de dados, proporcionando ao analista e projetista de banco de dados o esclarecimento necessário sobre os dados e suas relações.

3.5. Uma Notação para OOD

Booch [BOOCH90] propõe uma notação que combina quatro diagramas distintos para criar um projeto orientado a objeto, que são: um diagrama de classes que descreve as classes e suas relações; um diagrama de objetos que representa objetos específicos e as mensagens que passam entre elas; um diagrama de módulos ou diagrama de Booch que é usado para ilustrar os componentes de programa e o diagrama de processos que capacita ao projetista descrever como os processos são atribuídos a processadores específicos, dentro de um grande sistema.

Desenhar um diagrama não constitui análise ou projeto, uma vez que ele simplesmente captura o comportamento do sistema ou a visão e os detalhes da arquitetura. Daí a importância de se ter uma notação bem definida para o processo de desenvolvimento do *software*, não significando, porém, que qualquer aspecto será usado o tempo todo no sistema; e sim que ela é suficiente para expressar a semântica de grande parte da análise e projeto do sistema.

É interessante observar que cada detalhe é necessário para expressar a importância das decisões e os detalhes que facilitam a criação de ferramentas de engenharia reversa e fornecem a integração de *front end* e ferramentas CASE.

3.5.1. Modelos e Visões

As decisões de análise e projeto são capturadas considerando suas classes e objetos de acordo com a *view* física e a lógica, dinâmicas e estáticas; sendo que as duas dimensões são necessárias para especificar a estrutura e comportamento de um sistema OO. Daí a necessidade de múltiplas *views* para capturar todos

os detalhes de um sistema complexo, a fim de facilitar o entendimento do mesmo.

É necessário definir o número de diagramas para cada dimensão; pois cada diagrama representa a projeção para os modelos do sistema e mostra a herança de certas classes chaves e o fechamento transitivo de todas as classes usadas por uma classe particular. E neste sentido, os modelos do sistema denotam as classes e seus relacionamentos, além de outras entidades. Por exemplo, é impossível representar com somente um diagrama as classes e os relacionamentos de uma aplicação com mais de mil classes.

Todos os diagramas e entidades com mesmo nome e escopo são considerados referências para o mesmo item do modelo. Por exemplo, se a classe X aparece em dois diagramas diferentes, para o mesmo sistema, ambas as referências são para a mesma classe X.

3.5.2. Modelo Lógico vs Modelo Físico

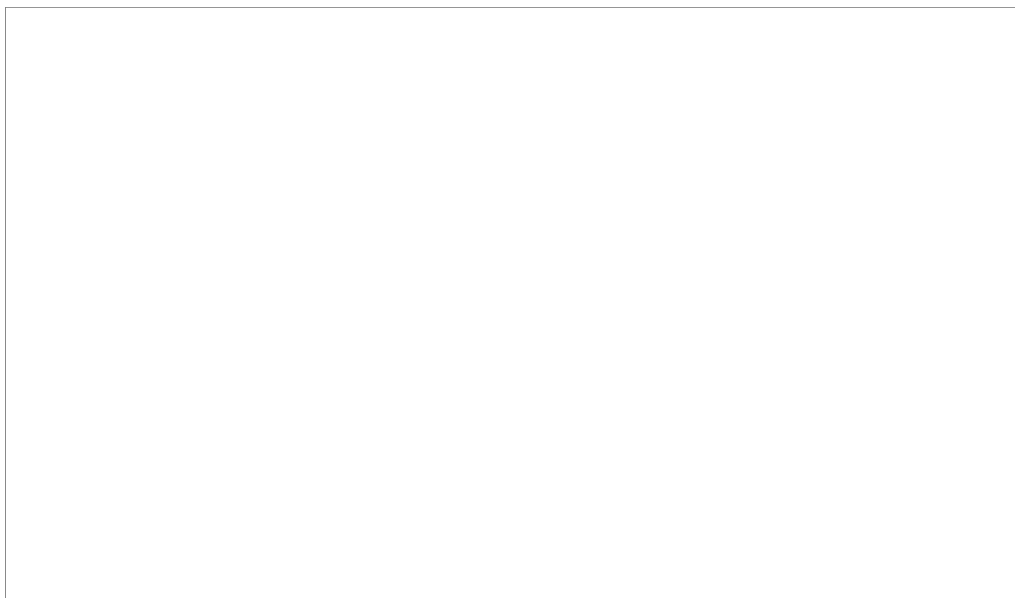
O modelo lógico serve para descrever a existência e significado das abstrações, assim como o mecanismo que forma o problema ou define a arquitetura do sistema. Já o modelo físico descreve a composição do *hardware* e *software* concreto ou a implementação.

No modelo lógico os primeiros meios para descrição dos cenários do sistema são os diagramas. Durante a análise usa-se diagramas de classe para capturar as abstrações dos objetos nos termos de regras comuns e responsabilidades.

3.5.3. Semânticas Estáticas vs Dinâmicas

Os diagramas de classe, objeto, módulo e processo introduzidos para resolver as questões da arquitetura do sistema estão longe de serem basicamente estáticos, pois objetos são criados e destruídos, enviam mensagens um para o outro, etc. Em desenvolvimento OO, as semânticas dinâmicas são expressadas por diagramas de transição de estado e de interação, onde cada classe tem um diagrama de transição de estado que indica o comportamento do evento das instâncias das classes.

3.5.4. Diagramas de Classe



Durante a análise este diagrama é usado para mostrar a existência de classes e seus relacionamentos no

modelo lógico do sistema e representar o modelo da estrutura de classe do mesmo; indicar as regras comuns e as responsabilidades das entidades que proporcionam o comportamento do sistema. E durante o projeto, é usado para capturar a estrutura de classes que formam a arquitetura do sistema.

Figura  Diagrama de Classe.

Para certos diagramas de classes é útil expor alguns atributos e operações associados a uma classe, pois um atributo denota uma parte do objeto usado durante a análise e expressa uma propriedade da classe, durante o projeto. E as operações denotam alguns serviços fornecidos pela classe, são nomeadas quando são mostradas no interior da classe e distinguidas dos atributos pela adição de parênteses. Assim, a sintaxe de atributos e operações podem ser adaptadas para as linguagens de implementação escolhidas.

As classes raramente ficam sozinhas, pois colaboram com outras, de várias maneiras como a associação, herança e os relacionamentos, sendo que cada relacionamento inclui um *label* que rotula um nome ou sugere seu propósito e cada associação liga duas classes e denota uma conexão semântica.

Durante o desenvolvimento, os relacionamentos são evoluídos através da declaração de conexões semânticas existentes entre duas classes e então são tomadas as decisões sobre a natureza exata destes relacionamentos.

A herança denota a generalização e especificação do relacionamento, aparecendo como uma associação com uma seta. A seta aponta para a superclasse e o outro lado da seta designa a subclasse. De acordo com a linguagem escolhida, a subclasse herda a estrutura e comportamento da superclasse.

3.5.5. Categorias das Classes

A classe é necessária, mas insuficiente como meio de decomposição. É necessário identificar os *clusters* das classes e suas coesões e para representá-los e particionar o modelo lógico do sistema utilizam-se as categorias das classes, que são agregações contendo classes e outras categorias de classes, no mesmo sentido que uma classe é uma agregação contendo operações e outras classes.



Muitas linguagens de programação não tem qualquer lingüística para suportar as categorias das classes e por esta razão, deve-se fornecer a notação para as categorias das classes que permitam expressar a importância do elemento que pode ser expressado diretamente pela linguagem de implementação.

Figura  Ícone Categoria de Classes [BOOCH94]

A figura acima mostra que o ícone pode ser usado para representar a categoria da classe; como para a classe em que o nome é requisitado para cada categoria da classe.

Algumas concepções são necessárias para expressar os detalhes de análise e projeto essenciais para o entendimento e visualização do sistema e estas se constituem de :

- Classes Parametrizadas que denotam uma família de classes de quem a estrutura e comportamento são definidos, independentes dos parâmetros formais da classe.
- Metaclasses que são classes de uma classe, sendo comum sua utilização para fornecer variáveis de instância da classe e suas operações.

- Classes Utilizáveis que podem ser parametrizadas e instanciadas em rotação e podem usar o mesmo relacionamento de instanciação que denote a relação entre uma classe parametrizada e sua instanciação.
- Aninhamento são classes que podem se aninhar fisicamente em outras classes e as categorias podem ser aninhadas em outras categorias. Corresponde, então, à declaração de aninhamento de uma entidade ocorrendo em um contexto fechado.
- Exportação de Controle onde os símbolos de acesso são aplicados para aninhar entidades em todos os seus formulários, podendo indicar acesso a atributos e operações.
- Restrições Físicas necessárias para a geração do código de projeto e da engenharia reversa.
- Regras e Chaves que ajudam na especificação do processo de análise e denotam a capacidade de uma classe se associar a outras.
- *Constraints* que são expressões de alguma condição semântica que deve ser preservada.

As especificações são formas que proporcionam uma definição completa de uma entidade na notação, como uma classe, uma associação e uma operação individual. Tem no mínimo, as entradas de nome que será um identificador e a definição, que será um texto descritivo representando a função da entidade.

Cada classe no modelo tem uma especificação da mesma com as entradas de responsabilidades, atributos, operações e *constraints*. E para cada operação que é membro da classe e todos os subprogramas, define-se uma especificação que fornece as entradas de retorno da classe e os argumentos.

3.5.6. Diagrama de Transição de Estados

É usado para mostrar o estado de uma determinada classe, os eventos ordenados do comportamento de um sistema como um todo, os que causam a transição de um estado para outro e as ações resultantes desta mudança de estado. Representa uma *view* de um modelo dinâmico de uma classe ou de todo o sistema.

Os dois elementos essenciais do diagrama de transição de estado são os estados e a transição de estados, onde o estado de um objeto representa os resultados cumulativos de seu comportamento e a transição de estado pode causar o estado de mudança do sistema. São insuficientes para descrever muitos tipos de sistemas complexos e por isto tem-se a notação abaixo:

- Ações Transições Condicionais de Estado, onde alguma ação pode ser especificada depois da palavra chave *entry* e *exit*. Geralmente, um determinado estado de transição terá um evento ou um evento e uma condição associado, ou ainda um evento não associado.
- Estados Aninhados abrangendo a explosão dos estados e as transições que ocorrem em sistemas complexos; pode ser aplicado em qualquer departamento e os subestados podem ser superestados de outros níveis mais baixos de subestados.
- História que ocorre quando se está transacionando de um estado com subestados e se quer retornar ao estado mais recentemente visitado.
- Estados Ortogonais que representam a decomposição de estados.



Em qualquer diagrama de transição de estado tem-se um estado *default* para o começo designado pela escrita de uma transição não rotulada para o estado de um ícone especial, mostrado como um círculo fino; sendo necessário, mais tarde do estado de parada; que é designado pelo desenho de um estado, também sem rótulo.

Figura 3.3- Diagrama de Transição de Estado para PRODUTO-VENDÁVEL:: status atual-estoque.[PAGE80]

3.5.7. Diagramas de Objetos

Este diagrama é usado para mostrar a existência de objetos e seus relacionamentos, em um projeto lógico; onde cada diagrama representa as interações ou relações estruturais que poderão ocorrer em um determinado conjunto de instâncias da classe e também uma *view* da estrutura de um objeto do sistema. Durante a análise, o diagrama é utilizado para indicar a semântica de cenários primários e secundários que fornecem um traço do comportamento do sistema; o que não acontece durante o projeto, pois é usado para ilustrar as semânticas dos mecanismos no projeto lógico do sistema.

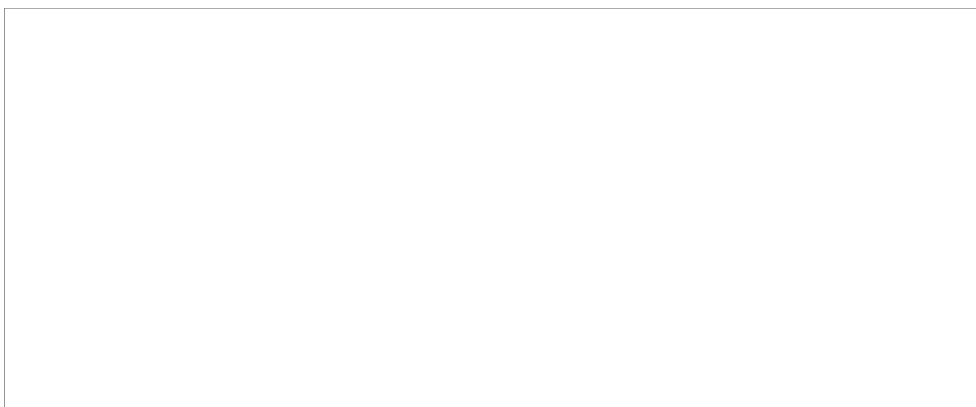


Figura 3.4- Diagrama de Objetos



Os elementos essenciais do diagrama são os objetos e seus relacionamentos, onde o relacionamento pode ser entre dois objetos com uma associação entre suas classes correspondentes, denotando um caminho de comunicação entre as instâncias da classe e um objeto poderá enviar mensagem para o outro.

A associação em diagrama de classe pode ser decorada com uma denotação de regra, o propósito ou capacidade que uma classe se associa com a outra, sendo, assim, útil para certos diagramas objeto na ligação entre dois objetos.

É importante observar o fluxo de dados das mensagens, facilitando, assim, a explanação de semânticas de um cenário particular; como também a visibilidade entre objetos, ou seja, como um objeto se torna visível para o outro. As associações denotam as dependências semânticas que podem existir entre as classes de dois objetos.

Certos objetos podem estar ativos, significando que eles personalizam suas linhas de controle ou podem ter somente semânticas sequenciais, garantindo que estas estejam presentes em múltiplas linhas de controle. Mas em cada uma destas circunstâncias tem-se duas saídas que são a representação dos objetos ativos que denotam as raízes de controle e as diferentes formas de sincronização entre cada objeto.

3.5.8. Diagramas de Interação

São usados para traçar a execução de um cenário do mesmo contexto, como um diagrama objeto. É simplesmente outra maneira de se representar o diagrama objeto, pois reestruturam os elementos essenciais do mesmo; sendo conceitualmente muito simples; são freqüentemente melhores que os diagramas objetos para capturar as semânticas dos cenários, no começo do desenvolvimento do sistema, antes dos protocolos das classes individuais serem identificados.

Existem dois elementos diretos que podem ser adicionados para fazê-los mais expressivos em certos projetos mais complicados, que são:

➤ *Scripts* que envolvem interações ou condições e podem ser escritos à esquerda de um diagrama de interação, usando uma forma livre ou um texto estruturado, ou ainda a sintaxe de uma linguagem de implementação.

➤ Controle do Foco indicando como as mensagens são passadas.

3.5.9. Diagramas de Módulos

São usados para mostrar a alocação de classes e objetos para o módulo no projeto físico, representando *views* para o mesmo.

Os dois elementos essenciais são os módulos e suas dependências, onde as dependências sugerem uma parte da compilação, como por exemplo, em C++ em que a diretiva *#include* indica a dependência de compilação.

Um sistema maior pode ser decomposto em muitos módulos que servem para particionar o modelo físico de um sistema em uma agregação, contendo outros módulos e outros subsistemas, podendo ser públicos e ou parte da implementação de subsistemas; mas por convenção, qualquer módulo em um subsistema é considerado público. Podem, ainda, ter dependências de outros subsistemas ou módulos.



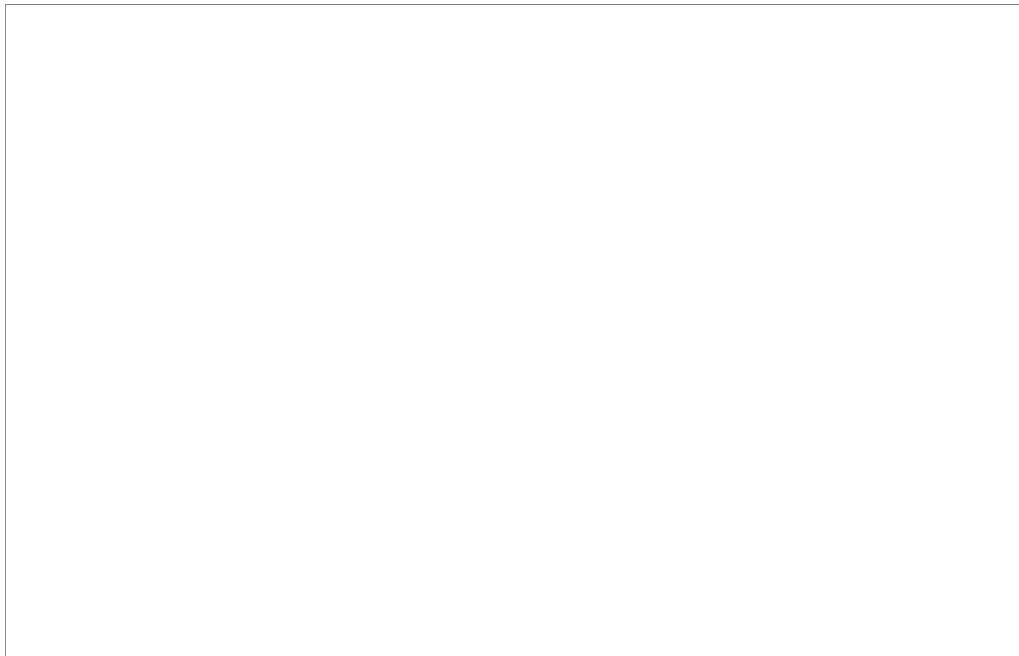
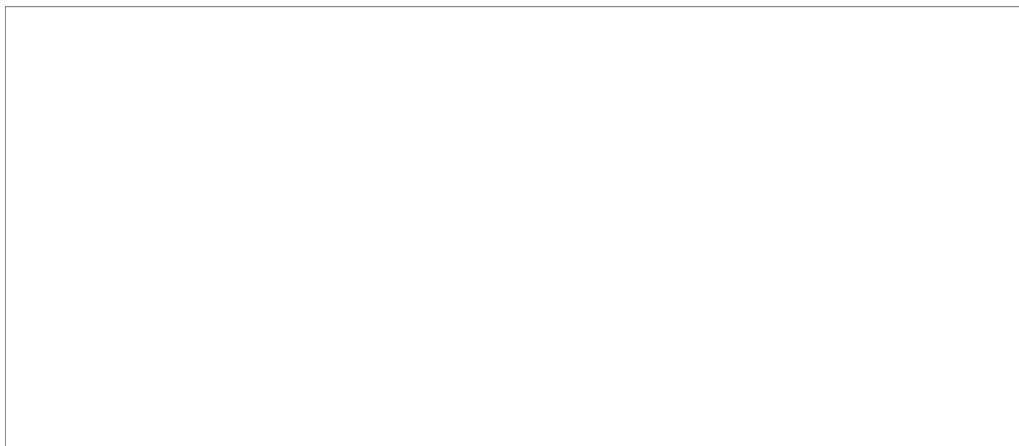


Figura 3.5.10 Notação Básica do Diagrama de Módulos [PRESSMAN95]

3.5.10. Diagramas de Processos



São usados para mostrar a alocação dos processos em um projeto físico; representam uma visão dentro da estrutura do processo; indicam a coleção física dos processos e dispositivos que servem como plataforma para a execução do sistema. Possuem três elementos essenciais que são os processadores, os dispositivos e suas conexões.

Figura 3.5.11 Notação Básica do Diagrama de Processos [PRESSMAN95]

O processador é um pedaço do *hardware* capaz de executar os programas; os dispositivos são pedaços do *hardware* incapazes de executar os programas e como os processadores, um nome é requisitado para cada dispositivo; as conexões representam algum acoplamento direto do *hardware* ou muitos acoplamentos indiretos, como por exemplo, satélites de comunicação.

3.6. Bibliotecas de Componentes Baseados em Objetos

Uma das vantagens de se ter biblioteca de classes reutilizáveis é que a reutilização pode ser concluída

pela extensão e especialização das classes encontradas nesta biblioteca.

3.7. Critérios Adicionais

Existem alguns critérios básicos de acoplamento e coesão para julgar a qualidade do projeto que são:

- A clareza do projeto é importante na ênfase da reutilização da OOD, ou seja, um projeto deve fazer sentido. Existem alguns fatores que contribuem para que o projeto seja claro, como: o uso de vocabulário consistente, a adesão a um protocolo de comportamento existente; evitar o excesso de modelos de mensagem e indefinições imprecisas de classes.
- Profundidade da Generalização- Especialização. A criação de níveis de classes de especialização resulta em uma hierarquia excessivamente profunda.
- Manter Objetos e Classes Simples através de quatro diretrizes: evitar atributos excessivos e muitos serviços por classe; focalizar as responsabilidades e minimizar a colaboração do objeto.
- Manter Protocolos e Serviços Simples . O vocabulário em um protocolo de mensagem deve ser tão simples quanto possível. Se uma mensagem requer mais de três parâmetros, alguma coisa está errada. Os serviços do OOD são pequenos, portanto se uma função é muito extensa, é preferível uma Estrutura Gen-Espec apropriada.
- Minimizar a Volatilidade do Projeto a fim de aumentar a qualidade do projeto. A volatilidade pode ser alta nas primeiras etapas do projeto, mas conforme a equipe experimenta diferentes projetos, discute idéias iniciais e aprimora gradualmente sua abordagem ela diminui.
- Minimizar o tamanho total do sistema evitando, assim, que ele alcance uma escala que exceda a habilidade da equipe de projeto, pois um sistema maior requer um número maior de classes , notação de OOD formal , documentação e critérios de avaliação
- Avaliação pelos “Fatores Críticos para o sucesso” que são a reutilização, legibilidade e performance.
- Elegância reconhecida no projeto com Repetições em padrões de Generalização – Especialização e colaboração e organização do projeto para refletir o domínio na prática.

3.8. Modificações e Benefícios de OOD

- Atacar os mais desafiadores domínios do problema, aperfeiçoando, assim, a interação entre especialistas em domínio do problema, analistas, projetistas e programadores e aumentando a consistência interna durante a análise, projeto e programação;
- Representar explicitamente elementos comuns;
- Construir sistemas resilientes a alterações;
- Reutilizar os resultados da OOA, OOD e OOP;
- Fornecer uma representação básica consistente para a OOA.

Alguns métodos proclamam que, para um determinado problema, há somente uma solução, ou seja, um projeto correto; e esse projeto não é apenas correto, mas ótimo. Mas nos sistemas complexos, o projetista é colocado entre alternativas, das quais deverá ser aplicada aquela que equilibra os componentes para minimizar o custo total do sistema por todo seu tempo de vida. E este tempo de vida envolve os custos de: análise, projeto e implementação; testes e depuração, *hardware* e os custos operacionais e de manutenção.

3.9. Métricas de Complexidade de Software Baseado em Objetos

Como os sistemas estão se tornando maiores e mais complexos, surgiu a necessidade de mecanismos para controlar essa complexidade, objetivando a diminuição de tempo e custo de desenvolvimento e o aumento da produtividade e manutenibilidade do *software*. Assim, muitos dos paradigmas de programação tem sido anunciadores da solução para o problema da complexidade e embora o paradigma orientado a procedimentos e a programação estruturada foram populares e seus fundamentos podem ainda ser válidos, a orientação a objetos é indicada como uma solução mais promissora para tempo e custo de desenvolvimento de *software*, pois quando usada efetivamente, permite a construção de aplicações manuteníveis e modificáveis; e o que é mais importante- satisfaz mudanças de requisitos do usuário.

Parâmetros são utilizados para medir as características específicas de um sistema, ou seja, métricas para gerenciar o processo de desenvolvimento. Assim, métricas são métodos quantitativos que auxiliam no desenvolvimento e aperfeiçoamento da qualidade e produtividade do *software*; visando a redução da complexidade, identificando as fraquezas do sistema e permitindo o aumento na qualidade, além de seu reuso e manutenibilidade.

A diversidade de métricas de complexidade de *software* é que permite ao desenvolvedor considerar os aspectos dos quais deseja medir em seu sistema, pois fornecem um *feedback*, evitando, assim, uma complexidade desnecessária; particularmente em projeto e programação OO, onde a complexidade se manifesta de muitas maneiras. Assim, elas podem ser divididas em medidas de produto e de processo; onde as medidas de produto visualizam o produto final, como por exemplo, Complexidade Ciclométrica de McCabe; e as medidas de processo medem o processo ou o ambiente usado para desenvolver e podem ser usadas de maneira preditiva, como o tamanho da aplicação.

3.9.1. Características das Métricas

As características das métricas são descritas a seguir:

- Algumas características principais da métricas são:
- Qualquer métrica é melhor que não ter métrica, pois não se pode gerenciar facilmente o que não se pode medir ;
- Geralmente são difíceis de serem coletadas;
- Tem valores diferentes para diferentes pedaços de *software*, não podendo se estender nos detalhes da implementação
- Não tem equivalência de interação e como a interação aumenta a complexidade, a métrica para a interação de dois pedaços de *software* poderá ser maior que a soma dos pedaços individuais; depende da ordem dos componentes do pedaço do *software*.

3.9.2. Tipos de Métricas

Os tipos de métricas são:

- Métricas Lingüísticas baseadas em características da linguagem de programação e o esforço para medir a complexidade nos números de operadores distintos;
- Métricas Estruturais que vêem o sistema em termos gráficos, como diagrama ou herança de rede e medem a complexidade com a estrutura dessa representação gráfica. As métricas para programação OO tendem a ser estrutural.

3.9.3. Métricas Orientadas a Objetos

Sistemas orientado a objetos exigem dois níveis de métricas que correspondem a dois tipos de estruturas presentes em cada sistema, que são:

- Métricas de nível de classes que permitem medir a complexidade de classes;
- Métricas para o nível do sistema que medem a complexidade das interações do objeto no projeto.

A tecnologia OO criou um novo desafio para os engenheiros de software, pois inclui novos conceitos como classes, métodos, mensagens, polimorfismo e encapsulamento; conceitos estes ausentes nas métricas tradicionais. Assim, métricas tradicionais nem sempre são úteis ao paradigma OO, fazendo surgir, então as métricas para sistemas OO.

3.9.4. Descrição das Métricas

As métricas descritas a seguir foram propostas por Chidamber e Kemerer [CHIDAMBER94].

3.9.4.1.Métodos Ponderados por Classe (MPC)

Estes métodos medem a complexidade de uma classe individual; são a soma da complexidade de todos os métodos por classe, onde o número e a complexidade de métodos envolvidos é um prognóstico de como tempo e esforço são requeridos para desenvolver e manter uma classe. Assim, classes com grande número de métodos tornam a aplicação mais específica, limitando a possibilidade de reuso da mesma.

3.9.4.2.Profundidade da Árvore de Herança (PAH)

Este método define a complexidade do grafo de herança de uma classe; é o número total de ancestrais para a classe; podendo, então, ser definido (e redefinido) em muitas classes. Assim, em casos de herança múltipla, o tamanho máximo de um nó é a raiz da árvore.

3.9.4.3.Número de Filhos (NF)

É o número de subclasses imediatas subordinadas à classe, na hierarquia de classes. Então, se a classe tem um grande número de filhos, poderá causar o mau uso das subclasses.

3.9.4.4.Acoplamento entre Objetos de Classes (AEO)

É o número de classes para as quais uma dada classe está acoplada; é uma relação clara entre métrica e abstração.

O grande número de acoplamento e a alta sensibilidade às mudanças em outras partes do projeto são mais difíceis e, por isso, mudar ou usar uma classe, sem entender todas as classes relacionadas é muito difícil. Assim como excessivos acoplamentos entre classes são prejudiciais na modularidade do projeto e evitam a reusabilidade.

3.9.4.5.Resposta para a Classe (RPC)

É o tamanho do conjunto resposta para a classe, onde este conjunto consiste em todos os métodos de classe, inclusive aqueles de outras classes chamadas pelos métodos das classes, enfim, é outra medida de acoplamento de classe.

Existem outros métodos de classes que podem ser descritos como métodos de classes que são: membros de classes, passados como argumentos e com instâncias que são criadas pelo método.

3.9.4.6.Falta de Coesão nos Métodos (FCM)

É o número de conjuntos desconexos formados pela interseção do conjunto de variáveis instanciadas, usadas por cada método da classe. Assim, a coesão de métodos da classe é desejável uma vez que promove o encapsulamento; o que indica que pouca coesão aumenta a probabilidade de erros, durante o processo de desenvolvimento.

Este método gera um número que resulta da subtração do número de pares de métodos sem variáveis de instância compartilhados e o número de pares de métodos com variáveis de instância compartilhados da classe, sendo, então, desejável, um valor alto para o mesmo.

3.9.4.7.Nomeando a Categoria (NC)

Este método divide as classes em um conjunto significativamente semântico, onde um método e uma variável de instância compartilhados estão relacionados desde que esta variável seja usada por este método. E dois métodos estão relacionados através da variável de instância, se ambos usarem aquela variável.

As métricas para *software* orientado a objeto apresentadas por Abreu e Carapuça [ABREU94] são definidas pela tabela, composta por vetores independentes, um contendo categoria (método, classe e sistema) e o outra granularidade (projeto, tamanho e complexidade). E as métricas formadas pelos elementos vetores são:

- Métrica para o Tamanho do Método;
- Métrica da Complexidade do Sistema.

Lorenz e Kidd [LORENZ94] subdividem:

- Métricas de Projeto para prever o tamanho da aplicação;
- Métricas de Desenho para examinar e melhorar a

qualidade do produto final , através dos componentes.

3.9.5. Métricas do Nível do Sistema

Estas métricas propõem outras métricas como o número de filhos medidos no escopo das propriedades de classes; número de hierarquia de classes como simples medida do sistema; número de *clusters* de classes medida do número de interconexões entre a classe no sistema e o número de *clusters* fundamentais na concepção do sistema. São definidas como o número de desconexões formados pela interseção do conjunto de classes associadas com cada classe.

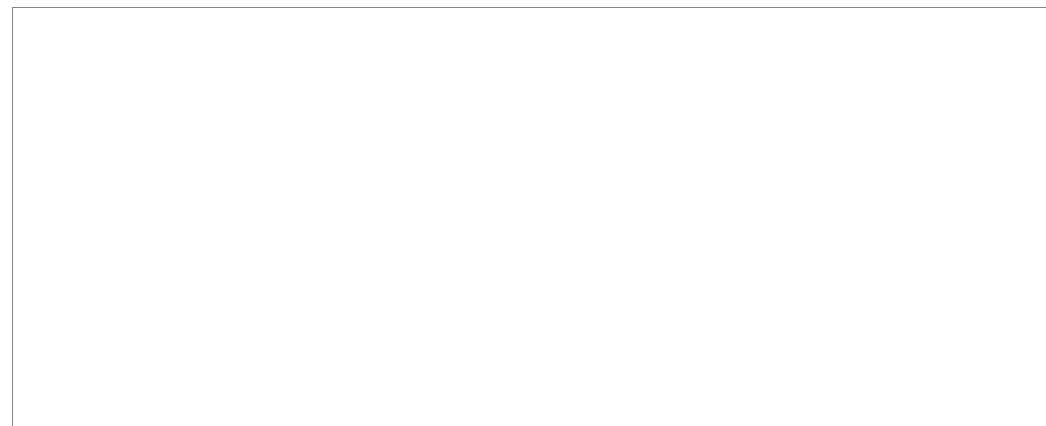
3.9.6. Complexidade da Associação

É medida da estrutura de associação da complexidade do sistema e é definida analogicamente da métrica de McCabe :

$$AC = A - C + 2P, \text{ onde :}$$

A - número de associações do diagrama.

C - número de classes no diagrama.



P - número de partes desconectadas no diagrama.

Figura 3.7 –Complexidade da Associação [KOLEWE93].

A figura acima mostra o cálculo desta complexidade.

3.9.7. Proposta de Implementação

Como a interação no sistema OO torna-se simplificado com introdução de novas classes, são utilizadas métricas como tentativas para medir a coesão e o acoplamento das classes e o nível do sistema. Algumas métricas são mais fáceis de coleccionar que outras, sendo fontes de informações para a tomada de decisão. Por isto, é importante identificar os caminhos propensos a falhas para que o teste / verificação possa se concentrar neles, evitando o gasto de tempo e recurso.

Pode-se também medir o *software* de forma dinâmica em tempo de execução e isto pode ser feito de duas formas:

➤ Medir a complexidade de caminho da mensagem baseada na Resposta para a Classe (RPC) e consiste em fazer o somatório dos valores atribuídos à complexidade de cada mensagem, enviada a um objeto, em tempo de execução. Para pontuar a complexidade de cada mensagem, deverá se preocupar com o número e tipo de mensagem enviada ao método.

➤ Preocupar-se com a complexidade das classes, as quais são enviadas mensagens em tempo de execução, tendo como base a Profundidade da Árvore de Herança (PAH), ou seja, quanto mais profunda a classe está na hierarquia, maior o número de métodos herdados e portanto, mais complexo será o acoplamento.

Concluindo, o *software* pode implicar em sistemas de grande ou pequena complexidade, encontrando fundamentos de tipos diferentes, pois algumas aplicações são largamente especificadas, construídas, mantidas e usadas, tendo muitos limites e geralmente são mais mais tediosas, que difíceis de desenvolver. Por isso, a funcionalidade de alguns sistemas é de difícil compreensão.

3.10. Conclusão

A tecnologia OO ainda está muito prejudicada pela falta de critérios de avaliação apropriados, pois há a introdução de conceitos novos, uma maneira nova de se utilizar componentes, assim como a reutilização dos mesmos, o que exigindo, assim, domínio para este novo paradigma por parte dos desenvolvedores. Mas isto não impede a adoção desta nova tecnologia pelas empresas, principalmente porque estas buscam soluções o problema da complexidade e falta de qualidade do *software*.

A complexidade do *software* é uma propriedade essencial, não acidental, derivando dos o domínio da complexidade do problema; a dificuldade do processo de desenvolvimento; a possível flexibilidade do *software* e os problemas característicos dos sistemas. Assim, é importante verificar os aspectos como as diferentes perspectivas de usuários e desenvolvedores na solução do problema; o domínio das técnicas que melhor se adaptam para a solução do mesmo, resultando, então em um sistema de qualidade que vise satisfazer as necessidades do usuário e não somente fazer com que o mesmo funcione.

Assim, existem várias medidas para redução da complexidade dos sistemas atuais, como as que foram utilizadas no paradigma procedimental e que algumas vezes podem ser úteis no novo paradigma OO e aquelas criadas para o novo paradigma, devido à introdução de novos conceitos.

Neste capítulo, portanto, espera-se que o leitor tenha condições de se identificar com este novo paradigma na construção de *softwares* mais flexíveis e reutilizáveis; além de sistemas construídos de maneira menos

complexa, através da aplicação de medidas e notações propostas.

4. Estratégias para Redução da Complexidade

4.1. Introdução

Desenvolver *software* orientado a objeto é difícil e fazê-lo reutilizável é ainda mais difícil. Pode-se encontrar objetos pertinentes, definir as interfaces das classes e hierarquias de heranças e estabelecer a chave das relações o que pode aumentar a complexidade do software, se as interfaces ou a hierarquia de heranças não forem bem definidas; e para isto, métricas para medir a complexidade e estimular a qualidade do *software* podem ser usados na fase de projeto, como, por exemplo, padrões de projeto que auxiliam a escolha de alternativas que tornam o sistema reutilizável, mas que não comprometem a reusabilidade. Pode, também prover a documentação e manutenção de sistemas existentes fornecidas pela especificação da classe e interações do objeto.

Cada padrão descreve o problema e o centro da solução. A reusabilidade é frequentemente um fator em projeto OO e as consequências incluem impactos no sistema, como flexibilidade, portabilidade e extensibilidade. E este capítulo a seguir busca justamente o estudo de métodos e técnicas que tornarão o sistema reutilizável e manutenível.

4.2. Diretrizes para Redução da Complexidade de Software Orientado a Objeto

A necessidade do controle da qualidade de *software* tomou duas formas que são o desenvolvimento de novos padrões de programação e medidas para monitorar a complexidade do código.

Segundo Booch [BOOCH94] os problemas quanto à limitação humana de lidar com a complexidade podem ser resolvidos através da otimização de desenvolvimento, aumento da produtividade e qualidade das tarefas do sistema e utilização da manutenibilidade e portabilidade.

É importante que se identifique ligações entre a complexidade e as características do sistema orientado a objeto e se defina formas para medir estas características, assim a complexidade pode ser medida através de métricas de complexidade estrutural. Estas métricas podem ser

➤ Intermodulares: medem estruturas de interconexões entre os módulos e caracterizam a complexidade a nível do sistema;

➤ Intramodulares: medidas pela complexidade procedural e semântica, sendo focadas a nível de módulo individual.

As métricas aplicadas ao OOD descritas a seguir estão melhor detalhadas no Capítulo 3. Aqui, serve somente de introdução para o assunto principal - estratégias para redução da complexidade.

4.2.1. Métricas Aplicadas ao Projeto Orientado a Objeto

4.2.1.1. Métricas Intermodulares (para complexidade do projeto do sistema)



Acoplamento entre Objetos (AEO) é o número de outras classes a que uma classe está acoplada, incluindo relacionamentos de herança, um baixo valor para esta métrica é o ideal.



Fan- In é o

grau de utilização do módulo, sendo aconselhável um alto valor para esta métrica.



Fan – Out

é o : número de módulos controlados por um módulo, está diretamente relacionado com a complexidade e portanto é importante que se tenha um valor baixo .



Fan –

Down é o número de subclasses que redefinem características da classe, profundidade na hierarquia de classes. Um valor baixo para esta métrica é o ideal.



Nível de

Hierarquia de Classes são as classes e as características a serem herdadas; exigindo maior entendimento para manutenção à medida que aumentam os níveis de hierarquia.



Largura da

Hierarquia de Classes é o número de subclasses diretas de uma classe.



Controle de

Passagem de Mensagem (CPM) é o quanto duas classes estão relacionadas , fazendo diferença na complexidade final.



Resposta

para a Classe (RPC) é o número de métodos usados por uma classe.



Parâmetros

por Método (PPM): poucos objetos como parâmetros para a mensagem, para não sobrecarregar o cliente.

4.2.1.2. Métricas Intramodulares (complexidade semântica)

A coesão semântica não pode ser medida , sendo um conceito distinguível, no estágio atual. Ela avalia o grau de abstração de uma classe.

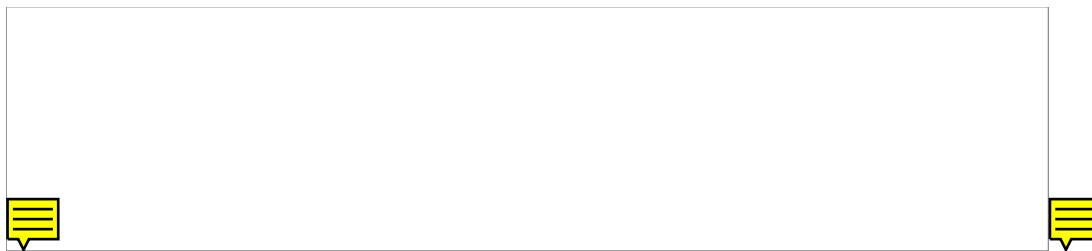
Uso de variáveis instância pelos métodos: inter-relacionamento dos métodos da classe e padrões das variáveis instância.

4.2.1.3. Métricas Intramodulares (complexidade procedural)

➤ Métricas de Tamanho são as medidas elevadas ou reduzidas de uma classe, não podendo usar tais medidas como determinantes numa subdivisão. Consistem no: número de classes do sistema; de métodos por classe; de variáveis de instância por classe; de classes herdando uma operação específica; medida da proporção do sistema de classes reutilizadas.

➤ Métricas da Estrutura Lógica representada pela Complexidade Ciclométrica de McCabe onde $(V(G))$ é a representação do módulo estruturado através do grafo acíclico direcionado (DAG). O valor de $(V(G))$ é melhorado (diminuído) quando usado o polimorfismo e a herança. Pode ser útil na divisão por métodos., sendo determinada pela análise do DAG junto com uma medida de coesão a nível do método. Para maiores detalhes sobre esta métrica, basta consultar o Capítulo 3.

4.2.2. Complexidade de Relacionamentos

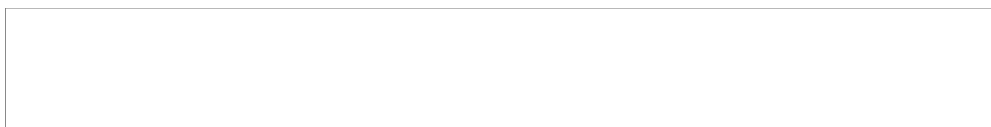


O acoplamento de herança está relacionado com a complexidade semântica da classe e os demais relacionamentos são considerados métricas intermódulos. A seguinte escala de acoplamento é verificada para demais casos de relacionamentos.

Figura 4.1- Escalas de Acoplamento [DIAS97].

O relacionamento de herança é o de maior acoplamento De forma geral, baixo grau de acoplamento é sempre desejável, mas durante o projeto, se um acoplamento maior entre as classes fornecer um menor grau de complexidade, pode-se dar preferência a ele. Porém, se a questão for distribuição e reusabilidade, deve-se utilizar o menor acoplamento possível, mesmo que haja acréscimo na complexidade interna da classe.

4.2.3.



Diretrizes para o Projeto OO

Figura 4.2- Desenvolvimento das Fases do Projeto [DIAS97].

É necessário que no plano de projeto se defina quais são as partes a serem trabalhadas uma de cada vez, de forma semelhante à determinação dos subsistemas.

4.2.3.1.No Projeto de Arquitetura



O projeto de arquitetura é desenvolvido de forma iterativa e de forma a resolver pequenas indecisões e realizar uma avaliação do trabalho, antes de partir para a próxima etapa.

Figura 4.3- Etapas da Fase do Projeto de Arquitetura [DIAS97].

➤ Decomposição em Sistemas procura os elementos de maior estabilidade até alcançar os menos estáveis, iniciando pelos componentes de Gerenciamento de Dados e Domínio do Problema, avançando para a Interação Humana e Gerenciamento de Tarefas.

A divisão por camadas poderá servir para a definição das partes de arquitetura, sendo fortemente relacionada com a reutilização e distribuição e refletindo principalmente na métrica *Fan-In*, o que não significa que outras métricas como a *Fan-Out*, CPM, RPC e PPM não serão afetadas.

➤ A Descrição de Subsistemas ajuda na descrição da comunicação entre os subsistemas, ou seja, quanto mais complicado for determinado tipo de componente, maior será a possibilidade do sistema não ser adequado.

A descrição da comunicação interna entre subsistemas é a elaboração de classes de fachada para a interface de um subsistema, a fim de reduzir o acoplamento entre os objetos e facilitar a comunicação do subsistema com outros.

4.2.3.2.No projeto Detalhado

a) Escolha de Mecanismos Globais estabelecer padronizações para possíveis soluções como esquemas de manipulação e persistência de coleções de objetos em memória.

b) O Ajuste de Estrutura de Classe: Descrição das classes e seus relacionamentos; ajustes e otimização do projeto. A nível de classe, esses ajustes podem ser refletidos para evitar o reprocessamento de expressões complicadas; replanejamento do processamento e criação de novas classes para melhorar alguma semântica.

No nível de relacionamento a abstração de características comuns, a reorganização de classes; a

substituição de herança quando esta não for semanticamente válida; a eliminação de heranças múltiplas, quando o ambiente não suportar; o acréscimo de associações e o projeto de implementação das associações.

McGregor e Sykes [MCGREGOR92] propõem o seguinte conjunto de diretrizes para a estrutura de classes:

- *Interface* pública de uma classe composta por métodos da classe;
- A classe deve esconder detalhes da implementação;
- Um operador deverá somente fazer parte da *interface* pública de uma classe se estiver disponível para usuários de instâncias da classe;
- Cada operador que pertença a uma classe, acessa ou modifica dados da classe;
- A classe deve ser independente;
- A interação entre duas classes deve envolver a passagem explícita de informação;
- Cada subclasse deve ser desenvolvida como especialização da superclasse com a interface pública da superclasse;
- A classe raiz deverá ser um modelo abstrato do conceito alvo;
- Classes reusáveis podem fazer uso de herança para modelar o relacionamento do domínio do problema;
- Número de métodos que entenderá a representação de dados da classe deverá ser limitado.

4.2.4. Eliminação da Herança Múltipla

A introdução de outras classes não está relacionadas com o domínio do problema, mas com o domínio da solução, podendo ser uma forma de eliminação da herança múltipla, redução da complexidade e acoplamento entre objetos (AEO), assim, é importante verificar a existência do problema de complexidade semântica com a herança múltipla.

4.2.5. Eliminação de Heranças Altas e Longas

Uma hierarquia é geralmente pouco profunda e muito larga, o que induz a um baixo grau de reutilização e busca de um novo nível de abstração, facilitando a compreensão do comportamento das classes.

4.2.6. Criando Associações Reutilizáveis

Associação do tipo $n \times n$ possui maior complexidade que do tipo 1×1 , pois os objetos com a primeira associação precisam controlar n ligações com outros objetos; por isto, é importante minimizar a complexidade das associações, tornando as diferenças menos acentuadas. E isto por ser conseguido através da criação de classes genéricas e / ou abstratas para que realizem o controle destas ligações.

Assim, a utilização de classes para associações facilita a transformação do diagrama de classes em código.

4.2.7. Empacotamento de Classes em Módulos

São pacotes de informações ocultas, com exceção das classes para interface, sendo que estas devem manter o mínimo de características visíveis.

A seguir, seguem-se alguns padrões de projeto que buscam a redução da complexidade, resultando em um sistema flexível, manutenível de qualidade.

4.3. Design Patterns

A concepção de padrões de projeto foi primeiro discutida por Christopher Alexander, em 1977 e talvez o primeiro livro a respeito tenha sido escrito em programas FORTRAN. E seu significado está na reusabilidade do código, que é uma importante saída para implicações financeiras e na identificação e descrição de soluções que especifiquem os problemas em forma de padrões.

Descrevem uma família de soluções para o problema do *software*; consistindo de muitos elementos de projeto semelhantes a módulos, *interfaces*, classes, objetos, métodos, funções, processos, relações entre objetos, etc. Tem comprovado utilidade em pequenos termos, ajudando no sucesso da reusabilidade. Enfim, é uma solução para o problema padrão; pois quando relacionado, formam a linguagem que provêem um processo para a resolução ordenada de problemas de desenvolvimento de *software*. Alguns exemplos são *Model / View / Controller* (MVC), *Blackboard*, Cliente Servidor e Controle de Processo.

Seu propósito é a captura do projeto a fim de torná-lo reutilizável; podendo, assim, melhorar a estrutura do *software* e a comunicação entre desenvolvedores e simplificando sua manutenção.

Linguagens padrões não são linguagens formais, mas particularmente uma coleção de padrões interrelacionados que, juntos, provêem um vocabulário sobre um problema particular. Auxiliam os desenvolvedores no conhecimento da comunicação; as pessoas a aprenderem novos paradigmas de projeto ou estilo de arquitetura.

Em geral muitas pessoas que estão trabalhando com padrões não estão concentradas no desenvolvimento de formalismos ou ferramentas para serem utilizadas junto com modelos; ao invés, elas estão concentradas na documentação de projetos, que é a chave para o sucesso dos desenvolvedores. Elas são motivadas por alguns valores como: sucesso é mais importante que a novidade; ênfase na escrita e clareza da comunicação; validação de qualidade do conhecimento; bons modelos aparecem com a experiência prática; identificar a importância da dimensão humana no desenvolvimento do *software*.

O primeiro *software* padrão foi escrito por desenvolvedores que trabalhavam com orientação a objeto e focaram no projeto e programação orientados a objetos ou na modelagem orientada a objeto. Embora haja muitos interessados em padrões orientados a objetos, uma nova tendência surge focada na eficiência, concorrência confiável e escalável, paralelismo e programação distribuída.

Muitos formatos de padrões usam notações populares como OMT e *Booch* para expressar suas estruturas e comportamento dinâmico.

Em poucos anos, os aspectos de padrões receberão considerável atenção com a integração de: padrões de projeto com *framework*; métodos de desenvolvimento e modelos de processos de *software* e para formar linguagens padrões.

Cada padrão de projeto foca-se em um problema de projeto orientado a objeto particular. São

baseados em soluções práticas que podem ser implementadas em linguagens de programação orientadas a objetos como *Smalltalk* e C++.

A essência de todos os padrões de projeto é o par problema-solução. Todas as outras partes do *template* adicionam os detalhes para a descrição do padrão(Figura 4.4).

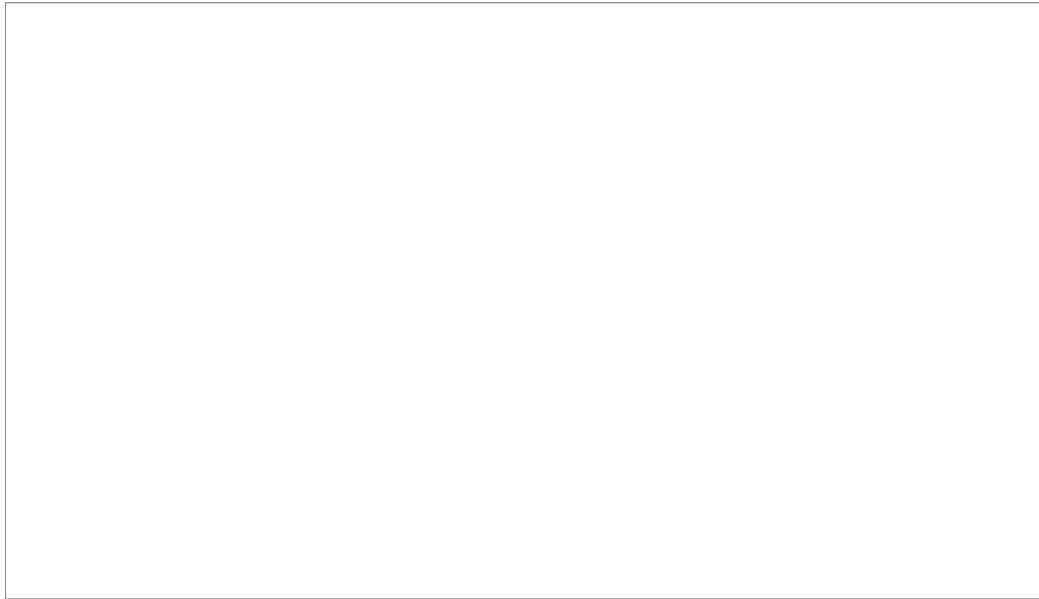
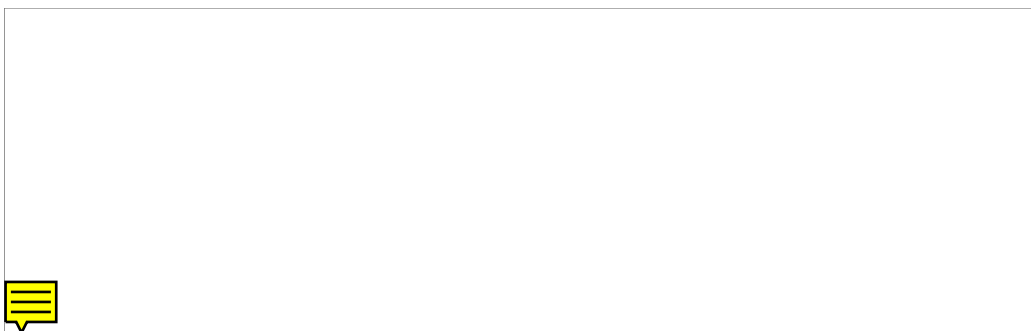


Figura 4.4- Padrões de Projeto compreendem um problema e uma solução que resolvam as forças contextuais em maneiras que as moldem para benefícios, conseqüências e outros padrões.

4.3.1. Padrões de Projeto em Smalltalk MVC:



Seu propósito é prover visões múltiplas e dinâmicas na distribuição de dados , mudando dinamicamente respostas para a entrada do usuário. Tem como flexibilidade a adição e remoção de visões, mudando, assim, a resposta ao usuário.

Figura 4.5 - O padrão é uma solução comum que foi reinventada em muitos contextos.

Sua implementação está na combinação de *observer*, *strategy* e *composite*.

Model / View / Controller (MVC) é uma tríade de classes usadas em interfaces com o usuário em *Smalltalk*, e consiste em três tipos de objetos, que são:



Model é uma aplicação objeto que se comunica com

as *views* quando há mudanças no modelo e estas *views* se comunicam estes modelos para acessar seus valores.

➤ *View* é a tela e assegura que a aparência refletirá no estado do modelo. Pode-se criar novas *views* para o modelo, sem reescreve-las. Utiliza uma instância do *Controller* para implementar uma estratégia particular ou diferente; ou simplesmente repor a instância com um tipo diferente de *Controller*.

➤ *Controller* define o tipo de interface com o usuário. Interface esta que consiste em aninhamento de *views* e que podem ser reusadas em um depurador. Suporta aninhamento de *views* com a classe *Composite View*, uma subclasse da *view*, que contém e gerencia o aninhamento de *views*.

O MVC muda a maneira de responder ao usuário, sem mudar o visual, como por exemplo, mudar a maneira de resposta ao teclado, ou usar *pop-up* menu, ao invés de chaves de comandos.

4.3.2. Descrições de Padrões de Projeto

4.3.2.1. Abstract Data Type (Classe)

O propósito é ocultar a estrutura de dados e acessar algoritmos entre mudanças não sensíveis de *interface*; com flexibilidade para mudar / repor implementação, I / O, sistema operacional, recursos de memória, sem afetar os clientes. Sua implementação está em fazer a *interface* independente de prováveis mudanças.

É possível criar múltiplas instâncias de ADT, mas somente um módulo e uma instância podem combinar vários relacionamentos de ADTs. Alguns exemplos são:

➤ *Repository* (Base de Dados) onde o propósito é prover uma estrutura de dados central com interface de acesso para múltiplos clientes. Clientes estes que são independentes, tendo, assim, a flexibilidade para adição e remoção dos mesmos e a implementação da estrutura de dados. Pode receber, estabelecer, questionar e atualizar métodos.

➤ Cliente Servidor com depósito distribuído, onde clientes e servidores rodam em computadores diferentes, conectados por alguma rede.

Gerenciador (Coleção) com funções semelhantes a criar / deletar, registrar, procurar, *layout* e *display* dentro da classe, separadas de objetos na coleção. Pode usar *Singleton* para o componente gerenciador.

4.3.2.2. Abstract Factory

Seu objetivo é prover uma interface para criação de famílias ou objetos dependentes, sem especificar suas classes concretas e empacotar construtores para a família de objetos relatados dentro de um objeto; sendo mais apropriado quando o número e tipos de objetos ficam constantes. Sua implementação está na *interface* comum do construtor através de famílias e múltiplas implementações.

4.3.2.3.Adapter

Converte a *interface* da classe com outra interface cliente, permitindo que as classes trabalhem juntas. Seu propósito consiste na conversão de uma determinada interface dentro de outra, adaptando-as.

4.3.2.4.Blackboard

Tem como propósito decidir dinamicamente que transformadores (conhecidos como origens) se aplicam na distribuição de estrutura de dados; podendo adicionar / repor transformadores e repor controlador. Tem os seguintes componentes: distribuição da estrutura de dados, conjunto de transformadores e controladores que selecionam transformadores.

4.3.2.5.Bridge

Desliga a união de sua implementação, permitindo que a abstração e implementação se desenvolvam separadamente, podendo adicionar novas abstrações ou implementação, sem afetar outras. Possibilita a separação de hierarquias de classes, para que estas trabalhem juntas e se desenvolvam independentes; a manutenção e o aumento das abstrações lógicas, sem tocar na dependência do código, e vice versa. Sua implementação consiste em separar camadas para abstração e implementação, além de fixar interface de implementação.

4.3.2.6.Broker

O propósito é encontrar servidores para requisições de clientes, adicionando / removendo servidores e clientes transparentemente através do protocolo para registrar servidores e serviços.

4.3.2.7.Builder

Separa a construção de um objeto complexo de sua representação, porém, alguns processos de construção podem ter diferentes representações. Especifica a criação do esqueleto usando construtores primitivos e usa a Delegação ao invés de *subclassing* para construção de primitivas.

4.3.2.8.Bureaucracy

Organizar uma estrutura hierárquica de objetos semelhantes que mantém a consistência interna, podendo estender a hierarquia e estender e transferir as responsabilidades. Sua implementação está na combinação de *Composite*, *Observer* e medida de responsabilidade.

4.3.2.9.Responsabilidade da Cadeia

Com o propósito de passar uma requisição para baixo da medida de objetos, evita o acoplamento do remetente da requisição para o destinatário. Sua flexibilidade está no desligamento e reconhecimento de uma requisição não conhecida a priori (determinada dinamicamente); na extensibilidade e na adição de novos reconhecimentos. Possui interface comum para reconhecedores.

4.3.2.10.Chamada a Procedure Remoto

Gerenciamento de Processo. Possui uma região condicional crítica (Mutex) chamada Semáforo. Esta região tem duplo fechamento checado; monitor; evento *loop*, *buffer* ilimitado, objeto ativo (futuro), *token* assíncrono, *half-sync* / *half-async*, transação e *rollback*, distribuição; resolvendo problemas que

aparecem em sistemas distribuídos com recursos distribuídos.

4.3.2.11.Command

Encapsula a requisição como um objeto, através da permissão de clientes parametrizados com diferentes requisições e determina uma interface uniforme para requisições que permitam a configuração de clientes; pode delegar todo, parte ou nenhuma das requisições de implementação para outros objetos; discute os mecanismos *undo* e *redo* construídos em interfaces e adiciona processador de comando. Possui flexibilidade para múltiplos comandos, adicionam funcionalidades como *undo / redo*, *scheduling*.

4.3.2.12.Composite

Tem o propósito de reconhecer partes / todo das hierarquias; clientes consideram múltiplos objetos atômicos e *composites* uniformemente. Possui, também, interface comum para *composites* e átomos em superclasse e múltiplas implementações.

4.3.2.13.Concorrência

Esta categoria resolve problemas em processamento paralelo.

4.3.2.14.Controle

Distribui com controles a execução e seleção de métodos certos, no tempo certo. É subdividido em Controle de Processo que tem como objetivo regular o processo físico, ajustar / repor o controlador. É implementado em controlador, processos variáveis, variáveis de entrada e manipuladas e sensores.

Suas variações são:

- *Open-loop-system* (processos variáveis não usados para ajustar o sistema).
- *Closed-loop-system*
- *Feedback control* (variáveis controladoras usadas para ajustar o sistema).
- *Feedforward control* (variável de entrada ou intermediária usada para ajustar o sistema).

4.3.2.15.Convenience Patterns

Simplificam o código. É subdividido em Método *Convenience* que simplifica o método pela redução do número de parâmetros; define métodos especializados, chamados por métodos gerais, fornecendo, assim, combinações de parâmetros.

4.3.2.16.Data Management

Empacota o estado dos objetos genericamente, independente do conteúdo atual dos objetos.

4.3.2.17.Decorator

Anexa responsabilidades adicionais para um objeto dinamicamente; fornece um alternativa flexível, estendendo funcionalidades, além de capturar o relacionamento classe- objeto que suporta o transparente “embelezamento”. O termo “embelezamento” refere-se a quaisquer responsabilidades para um objeto, como por exemplo uma árvore com sintaxe abstrata e ações semânticas, um estado finito autômato com novas transações, ou uma rede de objetos persistentes com *tags*.

4.3.2.18.Decoupling

O propósito deste padrão é dividir o software em partes, de maneira que as partes individuais possam ser construídas, mudadas e reusadas independentemente; sua vantagem é a mudança local através da modificação de uma, ou somente algumas partes do sistema, ao invés do todo.

Módulo, Abstração, *Data Type* e Camadas Hierárquicas tem larga aplicabilidade. *Iterator*, *Proxy*, *Facet* e *Visitor* se aplicam em situações de projeto restritas.

4.3.2.19.Estado

Permite que um objeto altere seu comportamento, quando seu estado interno mudar. Seu propósito é escolher o comportamento de acordo com o estado do objeto, adicionando / removendo estados. Pode ser implementado por formulários distribuídos para o estado finito e interface uniforme para ações. Implementa, ainda cada estado do objeto e comportamento apropriado para um estado determinado.

4.3.2.20.Evento Baseado na Integração

Permite que os objetos se comuniquem indiretamente, sem o conhecimento do outro, integrando, assim, objetos que não se conhecem. É implementado para registrar participantes em canais comuns que enviam eventos / dados para o canal.

4.3.2.21.Facade

Provê uma *interface* unificada para o conjunto de *interfaces*, no subsistema, ocultando alguns componentes. Pode mudar / repor subsistemas ocultos através da inclusão de *interfaces*.

4.3.2.22.Facet

Adiciona novas *interfaces* para classes existentes, sem mudá-las; provendo múltiplas *views*. Com flexibilidade para *interface* e funcionalidade estendidas, implementa registros de um objeto e os retorna, se questionado.

4.3.2.23.Flyweight

Usa a distribuição para suportar um número largo de objetos de grãos finos eficientemente, salvando espaço pela distribuição de estados comuns entre objetos. Sua implementação está em diferenciar entre estados intrínsecos e extrínsecos.

4.3.2.24.Framework

Conjunto de classes cooperando para fazer o projeto reutilizável para uma classe específica de *software* através da redefinição dos parâmetros do projeto, da captura de decisões do projeto e inclusão de subclasses concretas. Pode ser gerado para construir editores gráficos, para diferentes domínios ou compiladores para diferentes linguagens de programação; ou ainda, para uma aplicação particular. Assim, o *framework* oferece um mecanismo para que o gerente de projetos identifique quais atributos do *software* correspondentes à sua corretitude e desempenho funcionais são importantes e um meio para avaliar o progresso no processo de desenvolvimento em relação às metas de qualidade.

Seu propósito é prover uma camada da aplicação completa que poderá ser estendida pela subclasse, implementando subclasses, métodos; métodos *factory* ; *builders* e *factories* abstratos.

4.3.2.25.Gerenciamento da Variável

Objetos diferentes podem ser considerados uniformemente em algum programa.

4.3.2.26.Integração

Permite o desenvolvimento independente de componentes que trabalham juntos .

4.3.2.27.Iterator

Captura as técnicas para acesso e passagem de estruturas dos objetos; ilustra como o encapsulamento da concepção das variáveis ajuda na flexibilidade e reusabilidade; além de fornecer uma maneira para acessar os elementos de um objeto agregado seqüencialmente, sem expor a representação básica e proporcionar uma interface acessando componentes em um *aggregate* / *container*.

4.3.2.28.Máquinas Virtuais

Uma máquina virtual executa programas escritos em uma linguagem específica. Alguns exemplos são descritos a seguir:

- O Interpretador usa a representação para interpretar sentenças na linguagem, depositando componentes do programa e trabalhando com a memória , contador do programa e métodos para executar instruções na linguagem.
- O Emulador simula um processador de *hardware* no *software*, pois modificar o emulador é mais fácil que mudar o *hardware*.
- A Regra Baseada no Interpretador interpreta o conjunto regra, usando o fato base, com flexibilidade para repor o interpretador. Implementa um depósito de regras dos componentes, trabalhando com a memória, competidor de regra (ao invés do contador do programa) e a regra do interpretador.

4.3.2.29.Mediator

Define como um conjunto de objetos encapsulados se interagem; desprende o acoplamento pela

manutenção de objetos explicitamente. Sua implementação se resume em separar classes para objetos e *mediator*; objetos informam *mediator* de eventos significantes pela chamada direta ou através da notificação do evento.

4.3.2.30.Memento

Sem a violação do encapsulamento, captura e exterioriza o estado do objeto interno, salvando e restaurando – o . Sua implementação está em empacotar / desempacotar rotinas para exteriorizar / restaurar o estado.

4.3.2.31.Mestre / Escravo

Dinamicamente distribuídos, eles trabalham lado a lado com processos subordinados a protocolo, para o trabalho distribuído; podendo adicionar / remover escravo e escalar paralelismo. O processo mestre cria processos escravos, supre requisição e espera para completar ; podendo, então, destruir o escravo.

4.3.2.32.Método Factory

Define uma interface para a criação de um objeto, mas permite que a subclasse decida que classe instanciar; especifica a criação do esqueleto usando construtores primitivos, várias primitivas em subclasses.

4.3.2.33.Método Template

Define o esqueleto de um algoritmo em uma operação, cedendo alguns passos para as subclasses; permite que as subclasses redefinam certos passos de um algoritmo, sem mudar a estrutura do algoritmo; especifica o esqueleto do algoritmo usando primitivas, várias primitivas em subclasses ou pela delegação.

4.3.2.34.Módulo

É um grupo de componentes que mudam simultaneamente sem sensibilidade de *interfaces* não podendo ser instanciado mais de uma vez. Pode, então, mudar / repor a implementação, *hardware*, I / O, sistema operacional, recursos da memória , sem afetar os clientes, tornando a *interface* independente de prováveis mudanças. São suportados por *interfaces* Java, Modula, Ada e C.

4.3.2.35.Objeto Nulo (Stub)

Elimina freqüentemente testes para referências nulas, pela reposição a um objeto nulo, que é uma instância de uma classe com pseudo implementações dos métodos.

4.3.2.36.Pipeline (Pipes e Filtros)

Transmite dados através de uma seqüência de transformações independentes, repõem / adicionam / deletam estágios e mudam topologias; definem formatos, I / O; protocolos e adaptadores competindo com *pipes*.

4.3.2.37.Propagator

Propaga mudanças através da rede de objetos dependentes; estende / escolhe a rede e adiciona novas classes de objetos da rede. É implementado para registro uniforme e notificação da interface; registro direto ou através de evento. Suas subclasses são :

- *Strict Propagator* com / sem falha.
- *Lazy Propagator*.
- *Adaptative Propagator*.

4.3.2.38.Observer

Define uma para muitas dependências entre objetos. Porém quando um objeto muda de estado, todas as dependências são notificadas e atualizadas automaticamente. É um caso especial de *propagator* com somente um nível de dependência.

4.3.2.39.Protótipo

Especifica os tipos de objetos criados usando a instância prototípica e cria novos objetos pela cópia deste protótipo, mudando, assim, o protótipo.

4.3.2.40.Proxy

Fornece um substituto para outro objeto, a fim de controlar seu acesso; adicionar / retirar funcionalidades não planejadas transparentemente, sem afetar o objeto original ou os clientes sem subclasses; delega requisições para o original antes / depois de adicionar funcionalidades. Alguns exemplos são listados abaixo:

- *Decorator (Proxies Cascata)*.
- *Buffer Proxy, Cache Proxy*.
- *Logging Proxy, Counting Proxy* .
- *Firewall (Proteção do Proxy)*.
- *Sincronização do Proxy*.
- *Acesso Remoto ao Proxy*.

4.3.2.41.Recoverable Distributor

Seu objetivo é o estado replicado em sistema distribuído, mantendo a consistência e reabilitação e escolhendo a política de consistência e reconhecimento da falha. É implementado para gerenciar o estado local e global, recolher falhas locais e globais e atualizar os protocolos.

4.3.2.42.Singleton

Assegura que uma classe tenha somente uma instância e fornece seu ponto global de acesso, a fim de garantir a instância simples para a classe e registrar sua existência no membro estático.

4.3.2.43.Strategy

Define uma família de algoritmos, encapsulando cada um e fazendo as trocas e repondo / adicionando algoritmos usados. Sua implementação consiste na interface / superclasse comuns para diferentes algoritmos; múltiplas implementações de algoritmos e delegação de implementação. Um exemplo de é o relacionamento *View-Controller* que é um objeto que representa um algoritmo e é útil quando se quer encapsular estruturas de dados complexas.

4.3.2.44.Superclasse

Fornece a discussão uniforme de variáveis da classe pela localização da interface comum dentro da superclasse e adiciona novas subclasses e variáveis.

4.3.2.45.Visitor

Representa uma operação executada nos elementos da estrutura de um objeto; permite que se defina nova operação, sem mudar as classes dos elementos em cada operação, sendo apropriado quando se quer habilitar uma variedade de coisas diferentes para objetos que tem uma estrutura de classe fixa. Pode ser utilizado em operações de desligamento, existentes em variáveis para muitas classes; extensibilidade e adição de novas operações. Possui algumas variações como:

- *Default Visitor.*
- *Visitor Externo.*
- *Visitor Acíclico.*

4.3.3. Propagation Patterns

Propagation patterns são complementos de padrões de projeto descritos por Gamma [GAMMA94] e expressam grupos de objetos colaborando para um propósito específico; são executáveis e tem no mínimo duas aplicações: expressando *design patterns* para desenvolvimento de software orientado a objeto e usando padrões de projeto para guiar o projeto de *patterns propagation* e suas adaptações.

Padrões para problemas de programação orientada a objeto podem ser descritos usando notações de *Adaptive Programming* (AP). Um excelente padrão candidato é o *Visitor* que adquire aprimoramento e formalização concisa, usando adaptações.

O *Builder* pode ser implementado usando sentenças para descrever objetos de maneira robusta, sem referência à classe específica. Já o *Prototype* pode ser implementado usando uma especificação de passagem, para definir uma operação clone. A especificação da passagem define um subgráfico que necessita ser copiado. Em geral, AP é aplicável para padrões de projeto e envolve subgráficos selecionados de grandes gráficos. A capacidade de se adaptar também é útil para qualquer padrão que

envolva interações, desde que a especificação da passagem seja sucinta.

Muitos padrões de projeto para orientação a objeto são aplicáveis diretamente para AP. Por exemplo, o *Pattern Siemens Reflection* pode ser usado para implementar AP; ou o *Observer* é útil para separar modelos de *views* em AP. Em resumo, a idéia de padrão de projeto é útil para AP e o objetivo é prover abstrações que permitam expressar melhor estes padrões para OOP e AP, amenizando o efeito de muitas tarefas de projeto.

4.3.4. Patterns para Adaptive Programming (AP)

4.3.4.1. Adaptive Dynamic Subclassing

As idéias de AP têm sido reinventadas em diferentes domínios. E desde que a idéia tenha sido usada com sucesso, é tempo de se formular, em termos de padrões para isolar a chave das idéias e fazê-las mais fáceis.

Normalmente são usados cinco tipos de padrões para descrever AP, sendo o *Inventor's Paradox* o primeiro que usa muitos acoplamentos de forma livre. Este padrão tem quatro *subpatterns* chamados Estrutura *Shy Traversal*, Estrutura *Shy Objeto* e Contexto.

Estes padrões são úteis para o projeto do software, para projetistas que querem engrandecer suas técnicas e metodologistas que querem adicionar adaptações a seus métodos.

A Estrutura *Shy Traversal* e Contexto são aprimoramentos do padrão *Visitor*, sendo bons para descrever passagens e contextos, permitindo, assim, uma descrição elegante de objetos *visitor*. A Estrutura *Shy Traversal* distribui somente dois relacionamentos: passagens e classes de gráficos e a Estrutura *Shy Object* usa dois relacionamentos: representações do objeto textual e classes de gráficos estendidas. A Estrutura Contexto usa o comportamento e a modificação do comportamento.

4.3.4.2. Adaptive Builder

Aplicações necessitam de mecanismos para a construção de objetos complexos e independentes para as partes que criam os objetos e as ferramentas de Demeter e o Dicionário de Classes de Demeter particionam as descrições dos objetos, resolvendo, então, este problema.

Este método encontra requisições do padrão *Builder* que é o algoritmo para criação do objeto complexo, podendo ser independente das partes que fazem o objeto e da maneira que são reunidos. Assim, fazer uma aplicação robusta mecaniza a definição da classe gráfica e induz a operações semelhantes a *copying*, *displaying*, *printing*, *checking*, etc.

4.3.4.3. Classe Diagrama e Classe Dicionário

É independente do mecanismo que cria o objeto, sendo aplicável para qualquer sistema que necessite de uma ou várias classes gráficas. Pode mudar o tempo, sem afetar o processo. Usando este método, não há a necessidade de se definirem as classes *Director*, *Builder* e *ConcreteBuilder*.

4.3.4.4. Estrutura Shy Object

Seu objetivo é fazer as descrições do objeto, independentes dos nomes das classes e mudar a estrutura das classes. Durante a evolução de uma aplicação as estruturas da classe implicam em atualizações seus objetos. Assim, este modelo é útil em projetos orientadas a objetos para leitura e impressão de objetos, em notações definidas pelo usuário.

Um exemplo de projeto pobre que pode ser aprimorado pela Estrutura *Shy Object* é aquele que contém muitos construtores chamados para construir objetos *composite*.

É útil para o projeto de uma linguagem nova que tem controle sintático e necessita de implementar o aproveitamento das classes. Estende a aplicação da estrutura da classe, porém cada objeto conhece a análise e imprime a função.

4.3.4.5. Adaptive Interpreter

Determinada a linguagem, define-se, então, a representação para a gramática, será especificada pela classe recursiva dicionário; permitindo que um interpretador use a representação para as sentenças interpretadas na linguagem. Depois da criação da gramática, um interpretador é construído para interpretar sentenças da linguagem, invés de uma instância do problema.

Demeter usa esta gramática para criar uma parceria para a linguagem, que poderá particionar sentenças dentro das árvores do objeto e imprimir os objetos saída como sentenças na linguagem; onde cada sentença pode se dividir dentro de um único objeto.

O *Interpreter Adaptive* tem alguns benefícios como: suporte para a expansão da gramática, desde que a gramática esteja definida na classe dicionário (CD) facilitando sua modificação e para o interpretador da nova linguagem. O comportamento do interpretador é definido usando o *Propagation Pattern* (PP).

O uso da Estrutura *Shy Object* conduz a descrições mais robustas e curtas dos objetos, resultando em um teste fácil das estruturas da classe, antes do início da programação. Tem um efeito positivo no projeto da classe, pois permite testar suas estruturas para integridade na representação dos objetos. É útil na conjunção com o padrão *Prototype* e pode ser usado para criar um protótipo na estrutura *shy*.

4.3.4.6. Adaptive Visitor

Aprimora o padrão *Visitor*, representando uma operação que pode ser executada nos elementos da estrutura do objeto; reunião do código descrevendo a passagem com dependência mínima na estrutura da classe.

Para que uma operação seja executada na estrutura do objeto, muitos dos objetos envolvidos são casuais. Estes objetos tem uma simples passagem do comportamento e seus métodos são pequenos.

Adaptive Visitor adiciona novas operações facilmente, usando o poder do *Iterator* através de qualquer tipo da estrutura do objeto.

Os problemas de padrões para programação orientada a objeto poderão ser descritos usando as notações de AP e um candidato é o *Visitor*, que consegue um aprimoramento significativo ao usar a capacidade de se adaptar. Assim, as aplicações que queiram alterar dinamicamente a representação de um objeto, usam este modelo.

4.4. Conclusão

Experientes programadores de linguagens convencionais tem mostrado que experiência e conhecimento não estão organizados em torno de sintaxe, mas em estruturas como algoritmos e estruturas de dados.

É fundamental que se conheça os padrões e definições para as técnicas utilizadas, pois os mais experientes projetistas orientados a objetos as utilizam e um número cada vez maior de pessoas ouvem e aprendem sobre eles. Estes padrões fazem o sistema parecer menos complexo, devido ao seu alto nível de abstração e que fornecem soluções para problemas comuns. Enfim , são úteis em direção ao modelo de análise para o modelo de implementação.

Um dos problemas no desenvolvimento do *software* reutilizável é que freqüentemente ele tem que ser reorganizado ou restaurado, daí então a necessidade de padrões de projeto que determinam como reorganizar o projeto e reduzir a necessidade de restauração mais tarde.

É fácil ver o padrão, como solução, técnica que pode ser adaptada e reusada, mas é difícil ver quando isto será apropriado; caracterizar os problemas e as soluções; então, conhecendo seu propósito a escolha do padrão mais apropriado para determinado problema será mais fácil.

Em resumo, este capítulo mostra ao leitor o conhecimento de padrões e diretrizes para a construção de sistemas coesos e de complexidade reduzida; aumentando sua qualidade e facilitando sua manutenibilidade.

5. Conclusão

Com o avanço da tecnologia, a utilização de arquiteturas e sistemas operacionais poderosos e o aumento da complexidade das aplicações atuais, surgiu a necessidade de métricas e técnicas para a redução da complexidade dos sistemas; principalmente nesta época de competição no mercado de trabalho, onde tempo e custo de desenvolvimento fazem o diferencial. E a introdução de um novo paradigma espera ser uma grande solução para os problemas enfrentados por desenvolvedores e projetistas na construção destas aplicações.

A adoção do paradigma orientado a objeto procura ser a solução para os problemas de complexidade enfrentados pelos profissionais de informática, principalmente porque esta introduz novos conceitos que auxiliarão em todas as fases do desenvolvimento do *software*, diminuindo tempo e custo de desenvolvimento.

As métricas preocupam-se com caminhos mais complexos que necessitam de testes mais minuciosos, visando a redução da complexidade do sistema e a conseqüente integridade e qualidade do mesmo; sendo, então uma fonte crucial de informações para a tomada de decisões.

A tecnologia OO ainda está caminhando e enfrenta problemas como a falta de domínio dos profissionais e

falta de critérios de avaliação. Porém isto significa que ela seja ignorada por empresas e profissionais de informática, principalmente pelas diretrizes e medidas existentes que propõem sistemas manuteníveis, menos complexos e com maior qualidade.

Assim, este trabalho procurou mostrar alguns conceitos desta nova tecnologia orientada a objetos, assim como métricas, técnicas e padrões de projeto, que auxiliarão na construção de sistemas menos complexos; esperando ser o início de outros trabalhos que poderão ser desenvolvidos inclusive implementados em alguma linguagem OO como Ada, Eiffel ou *Smalltalk*; principalmente porque é um novo paradigma de desenvolvimento de *software* onde há muito o que descobrir, aprender e desenvolver.

Alguns dos trabalhos futuros que poderão ser desenvolvidos são:

- A implementação de um sistema utilizando as técnicas que auxiliem na redução de complexidade do sistema e outro que não utilize tais técnicas para demonstração de quão úteis poderá ser a adoção destas métricas e técnicas na vida do projetista e desenvolvedor.
- O desenvolvimento mais completo sobre design patterns e sua importância na melhoria da qualidade do *software*.
- A modelagem de um sistema Orientado a Objetos demonstrando onde a coesão, acoplamento e os conceitos da tecnologia Orientada a Objetos foi utilizada.

6. Referências Bibliográficas

[ABREU94] ABREU, F.B & CARAPUÇA R. ; “Candidate Metrics for

a Object Oriented Software within a taxonomy framework.”,

Journal of Systems and Software,1994,26(1): 87- 96.

[ALBRECHT79] ALBRECHT, A. J.; “Measuring Application

Development Productivity”, Proc.IBM Applic. Dev. Symposium,

Monterey, CA, Outubro 1979, pp. 83-

92.

[BOOCH90]BOOCH, G.; “Object Oriented Design”, Benjamin-

Cummings,

Califórnia,1990.

[BOOCH94] BOOCH, G.; “Object Oriented Analysis and Design with

Applications”,2 º edição, Benjamin-Cummings, Califórnia ,1994.

[CHIDAMBER94] CHIDAMBER. R. Shyam. & KEMERER. F. Chris.;

“ Towards a Metrics Suite for Object Oriented Design”,1994.

[COAD90]COAD P. & YOURDON E., “Object Oriented Analysis”,

Tradução CTInformática, Prentice Hall, 1990.

[DIAS97] DIAS, Márcio de Souza e ANDRADE, Renata; ”Diretrizes

para Redução da Complexidade de Software Orientada a Objetos”;

Disponível : [http:// www.ics. uci. edu / ~mdias/ public/cits97](http://www.ics.uci.edu/~mdias/public/cits97)

/cits97.html; acesso: 06/06/2000.

[ERICKSON00] ERICKSON, Tom; “The Interaction Design Patterns Page”; Disponível: [http://www.pliant.org /personal/ Tom_Erickson/ InteractionPatterns.html](http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html); acesso:26/06/2000.

[GAMMA94] GAMMA, Erich, HELM, Richard , JOHNSON, Ralph , VLISSIDES, John; “Design Patterns Elements of Reusable Object Oriented Software”, Addison Wesley, 1994, p. 1-358.

[HENDERSON96] HENDERSON, Sellers , B.; “Object Oriented Metrics: Measures of Complexity”, Prentice Hall , New Jersey, 1996.

[KIRSHBAUM00] KIRSHBAUM, David; “Introduction to Complex Systems Theory”; disponível :[http:// www.calresco.org/ intro.htm](http://www.calresco.org/intro.htm); acesso: 06/06/2000.

[KOLEWE93] KOLEWE,R.; “Metrics in Object Oriented Design and Programming”, publicado em Software Development, 1993.

[LARRY93] LARRY .Shiller.;“ Excelência em Software”, Tradução Equipe Flama Informática, Revisão Sílvio Carmo Palmieri, Makron Books, São Paulo,1994, p 119-126, 139-151,173-178,214-221,240.

[LORENZ94] LORENZ.M & KIDD.J.;“Object Oriented Software Metrics –A Pratical Guide”, Prentice Hall ,1994.

[MCCONNELL93] MCCONNELL, Steve., “Code Complete- A Pratical Handbook of Software Construction”, Microsoft Press,1993, p. 53-166; 399-551.

[MCGREGOR92] MCGREGOR ,J. D. & SIKES,D.A ; “Object Oriented Software Developmant –Engeneering Software for Reuse ” Van NostrandReinhold, New York, 1992.

[MEDEIROS00] MEDEIROS, Álvaro F.C.; “Tecnologia Orientada a Objetos”; Disponível: [http:// www.di.ufpb.br/ alvaro/cursos/ anal _](http://www.di.ufpb.br/alvaro/cursos/anal_)

2991/tsld045.htm ; acesso: 26/06/2000.

[MONTEIRO00] MONTEIRO, Antonio M.V. e CAMARA, Gilberto;

“Engenharia de Software”; Disponível: <http://www.dpi.inpe.br/cursos/cap365>; acesso: 26/06/2000.

[NORMAN90] NORMAN, Ron; “Object Oriented Analysis & Design”, 1990.

[PAGE80] PAGE JONES, Meilir, “Projeto Estruturado de Sistemas”,

Tradução Sílvia Maria Almeida Barros, Eliana Maria Leme Gotilla e Zileia Francisca dos Santos, Editora Makron Books, São Paulo, 1980, pp.63-78, 109-149.

[PRESSMAN95] PRESSMAN, Roger S. , “Engenharia de Software”,

Tradução José Carlos Barbosa dos Santos e Revisão Técnica Paulo César Masiero, José Carlos Maldonado e Rosely Sanches, 3° ed.; São Paulo; Makron Books, 1995; p. 54-86; 317-412; 415-484; 521-564; 723-833; 876-914.

[SICHONANY00] SICHONANY, Oni Reasilvia de Almeida Oliveira;

“Métricas para Complexidade de Sistemas Orientados a Objetos”; Disponível: http://www.inf.ufrgs.br/~oni/art_sem.html; acesso: 26/06/2000

[SHMIDT00] SHMIDT, Douglas C., JOHNSON Ralph E.; “Software

Patterns”; Disponível: <http://www.cs.wustl.edu/~schmidt/CACM-editorial.html> ; acesso: 26/06/2000

[TAYLOR59] TAYLOR, E.S.; “An Interim Report on Engineering

Design”, Massachusetts Institute of Technology, 1959.

[TICHY00] TICHY, Walter F.; “Essential Software Design Patterns”;

Disponível: <http://www.ipd.ira.uka.de/~tichy/patterns/overview.html>; acesso: 26/06/2000.

[YOURDON90] YOURDON, Edward e COAD, Peter; “Projeto Baseado

em Objetos”, Editora Campus; p 11-16; 125-146, 1990.

[YOURDON90]YOURDON, Edward e COAD, Peter; “Análise Baseada em Objetos”, Editora Campus, Caps. 1,4,5,7 P. 11-32; 25-31; 107-146,1990.