EMERSON PALACIO

APLICAÇÃO DE TÉCNICAS DE REUSO EM PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para conclusão do Curso de Especialização em Engenharia de Software MBA-USP

São Paulo
2003

EMERSON PALACIO

APLICAÇÃO DE TÉCNICAS DE REUSO EM PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para conclusão do Curso de Especialização em Engenharia de Software MBA-USP

Área de Concentração:
Engenharia de Software

Prof. Dr. Kechi Hirama

São Paulo
2003

Palacio, Emerson

Aplicação de Técnicas de Reuso em Processos de Desenvolvimento de Software.

São Paulo, 2003 72p.

Monografia (MBA- Engenharia de Software) Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia

AGRADECIMENTOS

Em especial à Sun Microsystems que possui uma excelente política desenvolvimento de funcionarios cujo o incentivo possibilitou a realização deste curso.

Ao Prof. Dr. Kechi Hirama, pelas diretrizes, orientações e constante incentivo que sem os quais não seria possível a execução deste trabalho.

A minha esposa Adrianne pelo seu constante apoio e incentivo em tudo que me proponho a fazer, além da infinita paciência e compreensão.

Aos meus colegas da Sun Microsystems Alexandre Akira e Celso Koyama que muitas vezes trabalharam além do horário nos meus meus plantões semanais possibilitando assim minha presença nas aulas.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Um dos objetivos da Engenharia de Software é o desenvolvimento de produtos de software com um grau de qualidade satisfatório. A constante busca de qualidade e produtividade no desenvolvimento levou a indústria de software a identificar novas soluções para melhorar o seu processo produtivo. Neste contexto, o processo de desenvolvimento baseado em reuso de software é muito importante, onde é possível identificar a forte relação entre os conceitos de qualidade e produtividade com o reuso de software.

As técnicas de reuso de software consistem basicamente no reaproveitamento de partes ou estruturas de software previamente desenvolvidas, qualificadas e armazenadas. Neste trabalho, apresentam-se os conceitos fundamentais do reuso de software no processo de desenvolvimento tais como: *patterns*, *frameworks*, engenharia e análise de domínio e também, uma análise dos modelos de ciclo de vida de software cascata e espiral como processos potenciais para a aplicação destes conceitos.

ABSTRACT

One of the purpose of the Software Engineering is the development of software products with a satisfactory degree of quality. The constant research of quality and productivity in the development takes the software industry to identify new solutions to improve the process productivity. In this context, the process of development that is based on software reuse is very important, where it is possible to identify the strong relation between the quality concepts and productivity with the software reuse.

The techniques of software reuse consist basically in taking the best advantage of parts or structures of software previously developed, qualified and stored. In this work, the basic concepts of software reuse in the development process are presented, such as: patterns, frameworks, engineering, domain analysis and also an analysis of the software life cycle models like waterfall and spiral as the potential processes for the application of these concepts.

1 – OBJETIVOS

Este trabalho tem como objetivo descrever processos de desenvolvimento de software baseado em reuso de software bem como conceitos e técnicas relacionadas ao reuso de software.

Entende-se que o reuso de software é aplicável no desenvolvimento de sistemas a partir de partes previamente desenvolvidas e armazenadas, ao invés de desenvolvê-las desde o início como nas abordagens convencionais.

1.1 - MOTIVAÇÕES

Entre os objetivos básicos da engenharia de software podemos destacar o desenvolvimento de produtos de software com um grau de qualidade adequada, e que o processo de desenvolvimento deste produto atinja a maior produtividade possível. É justamente neste contexto que se encaixa o processo de desenvolvimento baseado em reuso de software.

Nota-se, portanto uma forte relação entre os conceitos de qualidade e produtividade com o reuso de software. As indústrias de software não são mais regionais ou nacionais e sim globais, exigindo cada vez mais qualidade dos produtos e processos com alta produtividade para sua sobrevivência.

A busca de qualidade e produtividade no desenvolvimento levou a indústria a apresentar muitas soluções a fim de melhorar o processo nas áreas gerenciamento de processos de desenvolvimento, modelos de processos de desenvolvimento, métodos formais, métricas, métodos orientados a objetos, qualificação de software, ferramentas CASE, engenharia reversa entre outras.

O alto dinamismo da tecnologia nos dias atuais vem dificultando o desenvolvimento de software e soluções onde a rapidez de entrega juntamente com produtos de qualidade é cada vez mais exigida. A rápida evolução em termos de hardware, sistemas operacionais, linguagens e ferramentas de desenvolvimento faz com o que o reuso de software seja uma opção para se manter ágil com qualidade.

Apesar do assunto reuso de software não ser recente, a introdução deste conceito em uma organização encontra muitas vezes uma grande resistência cultural, porque representa aos desenvolvedores uma nova maneira de trabalhar.

Os ganhos de produtividade almejados com a pratica do reuso não são imediatos e nem em curto prazo. Desenvolver para o reuso de software requer uma série de mudanças técnicas e gerenciais na organização, que implicam em custos e tempo para que seus efeitos apareçam.

1.2 - ESTRUTURA DO TRABALHO

CAPÍTULO 1 – INTRODUÇÃO

No contexto deste trabalho será abordado o reuso de software baseado em análise de domínios, *patterns* e *framework* com ênfase em processos de desenvolvimento.

CAPÍTULO 2 - CONCEITOS DE REUSO DE SOFTWARE

Este capítulo aborda o reuso de software de uma forma conceitual, geral e ampla, passando por tópicos importantes como tipos de reuso, fatores de sucesso, maturidade, efeitos do reuso na qualidade e produtividade.

CAPÍTULO 3 - ANÁLISE DE DOMÍNIOS

Este capítulo aborda a definição de análise e arquitetura de domínio como conceitos chave para o reuso de software.

CAPÍTULO 4 - PATTERNS E FRAMEWORKS

Este capítulo aborda *patterns* e *frameworks* que são utilizados de maneira efetiva com as técnicas de reuso de software no processo de desenvolvimento.

CAPÍTULO 5 - PROCESSOS DE DESENVOLVIMENTO COM REUSO DE SOFTWARE

Este capítulo aborda os processos de desenvolvimento aplicando o reuso de software.

CAPÍTULO 6 – CONCLUSÕES

Este capítulo descreve as contribuições do trabalho e também aspectos importantes que podem ser desenvolvidos em trabalhos futuros.

CAPÍTULO 7 – REFERÊNCIAS BIBLIOGRÁFICAS

Neste capítulo são apresentados todos as referencias bibliográficas utilizadas neste trabalho.

2 - CONCEITOS DE REUSO DE SOFTWARE

A necessidade de reuso iniciou-se quando os seres humanos começaram a resolver problemas. Tentou-se solucionar problemas utilizando-se sempre soluções similares para novos problemas, onde muitas vezes havia um padrão de solução.

Já no desenvolvimento de software, para reduzir a quantidade de re-trabalho e aumentar a eficiência, deve-se ter a possibilidade de utilizar partes de trabalhos anteriormente desenvolvidos. Reuso de produtos de um ciclo de vida de software, significa poder reusar partes de um sistema desenvolvido anteriormente como: especificações, módulos de projeto, arquiteturas e código fonte (TERRY C, 1996).

Pesquisas das técnicas de reuso estão focadas no sentido de formalizar o reuso de software, porque é reconhecido que existe um aumento substancial da qualidade e produtividade com ganho econômico. O sucesso somente será alcançado se o reuso de software for conduzido formalmente e sistematicamente.

Segundo William Frakes (FRAKES, 1996), reuso de software é o uso de componentes de softwares existentes e ou conhecimento para a criação de um novo software, e o reuso também é o método para melhorar significantemente a qualidade e produtividade do software.

O tema reuso de software foi abordado na Conferência de Engenharia de Software em 1968 por Dough McIlroy, que comentou sobre componentes tornarem-se um ramo importante na engenharia de software, e ressaltou ainda a possível utilização de catálogos padronizados de rotinas, robustez e desempenho (MCILROY, 1968).

2.1 - REUSO DE SOFTWARE COMO ABORDAGEM SISTEMÁTICA

As organizações implementam o reuso sistemático de software para que seja possível medir o progresso e para identificar a estratégia de reuso de software mais efetiva. Esta sistemática deve ser bem planejada, formalizada com diretrizes, procedimentos estabelecidos e arquitetura definida de forma genérica e comum. A arquitetura comum captura características essenciais e possíveis variedades das aplicações a serem desenvolvidas. Outra característica da sistemática de reuso de software é a avaliação baseada em métricas, necessárias para estabelecer uma melhoria continua (FRAKES, W. e TERRY C, 1996).

O ganho da produtividade do reuso de software vem da integração prévia dos componentes escritos para um novo projeto de software. Pode-se dizer que o esforço do desenvolvimento é necessário para fazer um componente genérico de software para o reuso, entende-se assim que o desenvolvimento de componentes adequados para o reuso de software normalmente exige maior esforço e recursos do que um desenvolvimento de componentes customizados.

O sucesso com a técnica do desenvolvimento baseado em reuso de software será obtido somente se o beneficio do reuso de componentes for maior que o custo de implementação e manutenção do reuso. Entretanto, o beneficio será parcialmente medido em termos econômicos. Benefícios como, melhorar a qualidade e obter uma resposta imediata são qualitativos e dificilmente determinados com precisão (ROTHENBERGER,1999).

Outro fato importante é a avaliação do grau de reuso do software, que consiste em identificar no código fonte o grau de maturidade para o reuso, para isso existem modelos de avaliação de maturidade. Este modelo é similar ao modelo de capacidade de maturidade desenvolvido pelo SEI - Software Eng. Institute da Carnegie Mellon University (HUMPHREY, 1988). O modelo de maturidade está no plano central do reuso de software, ajudando as organizações a entenderem o passado, presente e objetivo futuro referente às atividades de reuso de software. Muitos modelos de

maturidade para reuso de software tem sido desenvolvidos e usados embora não tenham sido validados (FRAKES, W. e TERRY, C., 1996).

2.2 - MODELO DE CAPACITAÇÃO DE REUSO

O SPC - Software Productivity Consortium desenvolveu um Modelo de Capacidade de Reuso que possui dois importantes módulos: o módulo de avaliação e módulo de implementação.

O módulo de avaliação consiste em um conjunto de fatores críticos de sucesso para a efetivação do reuso de software, declarados como objetivos onde uma organização pode avaliar o estado presente da prática de reuso de software. Os fatores são organizados em quatro grupos primários: gerenciamento, desenvolvimento da aplicação, conjunto de desenvolvimento e processos / fatores de tecnologia (DAVIS, 1993).

O módulo de implementação ajuda a priorizar objetivos e a desenvolver com êxito os estágios de implementação do reuso de software. O SPC identifica quatro estágios para reduzir o risco do reuso e aumentar a implementação do Modelo de Capacidade de Reuso:

Oportunista: a estratégia de reuso de software é desenvolvida no nível do projeto, ferramentas especializadas de reuso de software são utilizadas onde conjuntos componentes são identificados.

Integrado: a estratégia do reuso de software é definida e integrada no processo de desenvolvimento de software da organização. Este processo de reuso é completamente apoiado pela gerência e pela equipe de desenvolvimento. Neste processo os componentes são categorizados.

Alavancado: esta estratégia de reuso de software expande o ciclo e vida e é especializada para cada linha do produto. O desempenho do reuso de software é medido e a fragilidade do software é identificada.

Antecipado: novos ciclos de negócios melhoram a capacidade de reuso de software e componentes, onde o retorno de investimento de componentes são identificados e as tecnologias de reuso de software são gerenciadas pelas necessidades dos clientes.

2.3 - REUSABILIDADE

Qualquer discussão sobre o recurso de software seria incompleta sem o reconhecimento da reusabilidade (FREEMAN, 1987). A maioria dos observadores da indústria de software concorda que a melhora na qualidade do produto e na produtividade do desenvolvimento de software está trazendo o fim da crise do software que muitas vezes assola o software. Em tal mundo, o software preparado para o reuso seria abundante (TRACZ, 1988). Existem bibliotecas de reuso de software para aplicações comerciais, para sistemas de tempo real e para problemas científicos e de engenharia. Porém, há poucas técnicas sistemáticas para se fazer adições em uma biblioteca, interfaces padrões para software reutilizável são difíceis de ser impostas, questões de qualidade e manutenibilidade permanecem sem solução.

Duas regras devem ser consideradas pelo gestor de software quando o software para reuso for especificado como um recurso (PRESSMAN, 1995):

1. Se o software existente cumprir os requisitos, adquira-o. O custo de aquisição de um software existente quase sempre será menor do que o custo para desenvolver um software equivalente.

2. Se o software existente exigir alguma modificação antes que possa ser adequadamente integrado ao sistema, proceda cautelosamente. O custo para modificar um software existente às vezes pode ser maior do que o custo para o desenvolver um software equivalente.

Ironicamente, os recursos de software freqüentemente são negligenciados durante o planejamento, só tornando-se uma preocupação suprema durante a fase de desenvolvimento do processo de engenharia de software. É bem melhor especificar cedo as exigências de recursos de software. Dessa forma, a avaliação técnica das

alternativas poderá ser realizada e a aquisição poderá ocorrer em tempo oportuno (PRESSMAN, 1995).

2.4 - COMPONENTES DE SOFTWARE

O software é uma informação que existe em duas formas básicas: componentes executáveis e não executáveis em máquina. Para o propósito do trabalho, somente os componentes de software que levam diretamente as instruções executáveis em máquina serão apresentado.

Os componentes de software são criados por meio de uma série de conversões que mapeiam as exigências do cliente para código executável em máquina. Um modelo (ou protótipo) dos requisitos é convertido num projeto. O projeto de software é convertido numa forma de linguagem que especifica a estrutura de dados do software, os atributos procedimentais e os requisitos relacionados. A forma de linguagem é processada por um tradutor a que converte em instruções executáveis em máquina (PRESSMAN, 1995).

O fato de poder reusar é uma característica importante de um componente de software de alta qualidade (BIGGERSTAFF, T.J, 1984). Ou seja, o componente deve ser projetado e implementado de forma que possa ser reusado em muitos programas diferentes. Na década de 1960, construíram-se bibliotecas de sub-rotinas científicas que eram preparadas para o reuso num amplo conjunto de aplicações científicas e de engenharia. Essas bibliotecas de sub-rotinas reusavam algoritmos bem definidos, mas tinham um domínio de aplicação limitado. Atualmente, a visão do reuso foi ampliada a fim de envolver não só algoritmos, mas também estruturas de dados. Um componente da década de 1990 engloba tanto dados como processamento num único pacote (às vezes chamado *classe* ou *objeto*), possibilitando que um engenheiro de software crie novas aplicações a partir de partes reusáveis. Por exemplo, as interfaces interativas de hoje freqüentemente são construídas utilizando-se componentes reusáveis que possibilitam a criação de janelas gráficas, menus do tipo *pull-down* e uma ampla variedade de mecanismos de interação. As estruturas de dados e detalhes de processamento exigidos para se construir a interface com os usuários estão contidas

numa biblioteca de componentes reusáveis para a construção de interface (BIGGERSTAFF, T.J, 1984).

Os componentes de software são construídos usando uma linguagem de programação que tem um vocabulário limitado, uma gramática explicitamente definida e regras de sintaxe e semântica bem formadas. Esses atributos são essenciais para a tradução por máquina. As formas de linguagem em uso são linguagens de máquina, linguagens de alto nível e linguagens não-procedimentais.

A linguagem de máquina é uma representação simbólica do conjunto de instruções da unidade central de processamento (CPU). Quando um bom desenvolvedor de software produz um programa bem documentado, capaz de sofrer manutenção, a linguagem de máquina pode fazer um uso extremamente eficiente da memória e otimizar a velocidade da execução do programa. Quando um programa é projetado pobremente e tem pouca documentação, a linguagem de máquina não contribui pra a solução.

No decorrer da última década, um grupo de linguagens de quarta geração, ou *não-procedimentais*, foi introduzido. Em vez de exigir que o desenvolvedor de software especifique detalhes de procedimento, a linguagem não-procedimental assume um programa especificando o resultado desejado, em vez de especificar a ação exigida para se conseguir esse resultado (COBB, 1985).

O software de apoio converte a especificação do resultado num programa executável em máquina. Até hoje, as linguagens de quarta geração tem sido usadas em aplicações de bancos de dados e outras áreas de processamento de dados comerciais (PRESSMAN, 1995).

Um componente de software pode ser uma estrutura de dados (ou banco de dados) ou um programa ou um componente procedimental (isto é, um módulo). Em cada caso, o componente de software deve ser projetado de tal maneira que possa ser reusado sem um conhecimento detalhado de seu funcionamento interno.

2.5 - TIPOS DE REUSO

No processo de desenvolvimento de sistemas, pode se aplicar o reuso de software em vários momentos. Existe a possibilidade de se reusar idéias, especificações, projetos, códigos-fonte e outros produtos nas diversas fases do processo de desenvolvimento, conforme pode ser visto na tabela 1 abaixo:

Substância	Escopo	Modo	Técnica	Intenção	Produto
Idéias	Vertical	Planejado	Composicional	Caixa Preta	Códigos-fonte
Artefatos	Horizontal	Ad-hoc	Gerativo	Caixa branca	Projetos
Procedimentos, Habilidades					Especificações
					Objetos
					Textos

Tabela 1 - Perspectivas de reuso de software (PRIETO-DIAZ, 1993).

2.5.1 - REUSO DE IDÉIAS

Este tipo de reuso envolve a utilização de conceitos formais, como soluções genéricas para uma classe de problemas. Um dos melhores exemplos é através dos algoritmos genéricos, como os desenvolvidos por Donald Knuth (PRIETO-DÍAZ, 1996).

O estado de arte dos algoritmos reusáveis, tanto na pesquisa, como na prática é relativamente maduro. Porém, pela perspectiva do reuso de software, são necessários o desenvolvimento e a distribuição de catálogos com informações específicas e detalhadas para a reuso desses algoritmos.

2.5.2 - REUSO DE ARTEFATOS

Este é o tipo de reuso onde o foco sempre esteve presente. Fábricas de software e conjunto de projetos corporativos sempre procuraram repetir as experiências da empresa de Raytheon dos EUA (McCLURE, C., 1992) e de outras empresas que conseguiram obter sucesso.

Atualmente, as técnicas orientadas a objetos foram vistas como a maneira mais promissora de reuso de software. Ambientes como Eiffel e ferramentas como Motif vêm com bibliotecas de objetos integrados.

O reuso de artefatos está concentrado na qualidade e na confiabilidade. A adaptabilidade dos componentes também é considerada um assunto importante neste contexto. Coleções de componentes específicos para um domínio prometem ser uma nova tendência neste tipo de reuso.

2.5.3 - REUSO DE PROCEDIMENTOS

As pesquisas de automação de processos e de ambiente centrado em processos têm dado foco na formalização e encapsulamento de procedimentos de desenvolvimento de sistemas. A comunidade de pesquisa sobre o reuso tem estudado a possibilidade de criar repositórios de procedimentos para reuso que podem ser interconectados entre si, criando processos mais complexos. A reutilização de processos é uma das atividades do projeto de Tecnologia de Software para Sistemas Adaptáveis Confiáveis (Software Technology for Adaptable, Reliable Systems – STARS) dos USA. (JACOBSON, 1997).

O reuso de procedimentos significa também, o reuso de habilidades e de conhecimentos. Esta área tem recebido uma atenção significativa da comunidade de sistemas especialistas, mas pouco da comunidade que trabalha com o reuso. Os gerentes de projetos tendem a reusar as habilidades informalmente, quando alocam as pessoas, mas não existe um esforço formal para capturar e encapsular os

conhecimentos destas, para torná-las parte do conhecimento da organização (PRIETO-DIAZ, 1993).

2.5.4 - REUSO VERTICAL

Reuso vertical é o que ocorre dentro de um mesmo domínio ou de uma área de aplicação. O seu objetivo é derivar modelos genéricos para famílias de sistemas que podem ser utilizados como gabaritos para a criação de novos sistemas. Quanto mais restrito for o domínio, maior será o retorno (PRIETO-DÍAZ, 1993).

Na grande maioria das vezes as fábricas de software são construídas para trabalhar com o reuso vertical. As organizações com projetos de longo prazo, como a National Administration of Space and Aeronautics NASA dos EUA concentram-se neste tipo de reuso.

O foco do reuso vertical é a análise e a modelagem de domínio que será abordado no capítulo 3. Pode-se citar dois métodos, o Feature Oriented Domain Analysis (FODA) e também o Knowledge Acquisition for Perservation of Trade-Offs and Underlying Rationales (KAPTUR), que são bastante populares e utilizados pela comunidade. Métodos orientados a objetos foram também estendidos para cobrir a análise e a implementação de domínio. Uma tendência atual neste sentido é a utilização de padrões de projeto (*patterns*, que será abordado no capitulo 4), que consistem em modelos de objetos para utilização dentro de um certo domínio (PRIETO-DÍAZ, 1996).

2.5.5 - REUSO HORIZONTAL

O reuso horizontal tem, como meta, a utilização de partes dentro de diferentes aplicações. Sub-rotinas de bibliotecas científicas e ferramentas do sistema operacional UNIX são bons exemplos de elementos deste tipo de reuso. A sua característica é a construção de componentes que podem ser utilizados em diversos domínios (PRIETO-DÍAZ, 1993).

A criação de componentes genéricos pode ter um custo maior, dada a necessidade de se implementar a flexibilidade necessária para a sua utilização em muitos domínios.

Embora as pesquisas de reuso horizontal tenham se concentrado em classificação e empacotamento de partes, o foco atual é o desenvolvimento de bibliotecas com amplo acesso (*broad-access libraries*). A construção de redes de repositórios interoperáveis e mecanismos mais eficientes de catalogação e de classificação estão causando um progresso significativo nesta área.

2.5.6 - REUSO PLANEJADO

O reuso planejado é a prática sistemática e formal do reuso onde diretrizes e procedimentos são definidos e seguidos, e métricas são utilizadas para verificar o seu desempenho. É uma prática normalmente adotada nas fábricas de software (PRIETO-DÍAZ, 1993).

É importante saber que o reuso planejado requer um alto grau de investimento e comprometimento gerencial. Ela requer uma mudança significativa na prática de desenvolvimento de software, principalmente cultural que implica na disciplina e no comprometimento de todos os envolvidos no processo de desenvolvimento.

Com o objetivo de guiar as empresas que pretendem adotar esta prática de reuso planejado, foram desenvolvidos vários modelos de maturidade, na sua maioria inspirados nos níveis de maturidade do modelo de maturidade de software (Capability Maturity Model – CMM) (FRAKES, 1996).

2.5.7 - REUSO AD-HOC

O reuso Ad-hoc é conhecido como reuso informal, em que os componentes são escolhidos de uma maneira não sistemática. O reuso é conduzido pelo desenvolvedor e não pelo projeto, e não há processos definidos para a utilização desta abordagem. Muitas vezes este tipo de reuso é chamado de oportunista (PRIETO-DÍAZ, 1993).

Este método está baseado na idéia de como o sistema de software é desenvolvido, pode-se parar no estágio do ciclo de vida escolhido, consultar a biblioteca de reuso e encontrar algo que se pode reusar. Embora proveitoso, o reuso ad-hoc é

economicamente limitado. Nas bibliotecas, os componentes são tão específicos e limitados que a adaptação a um novo conjunto de requisitos é normalmente muito oneroso e improdutivo (PRIETO-DÍAZ, 1993).

O desenvolvimento da melhoria e capacitação das bibliotecas de reuso têm sido a ênfase do reuso ad-hoc. Entretanto, o atraso no desenvolvimento ocorre devido à catalogação e classificação de componentes, que são manualmente elaborados. Apesar do progresso já alcançado, é necessário que se pesquise mais sobre a tecnologia das bibliotecas de reuso.

Como este modelo é o mais conhecido de reuso, existem investimentos para torná-lo mais eficiente. O principal progresso nesta área é o avanço tecnológico das ferramentas de catalogação e de busca de componentes. Porém, a principal barreira para sair desta forma de reuso é cultural, sendo necessário mudar a postura e a maneira de trabalho dos desenvolvedores de software (PRIETO-DÍAZ, 1996 e 1993).

2.5.8 - REUSO COMPOSICIONAL

O reuso composicional é a utilização de componentes existentes como blocos para a construção de um novo sistema. É baseada na utilização de coleções de reuso, de bibliotecas de sistemas e de padrões de interfaces. Embora os defensores desta abordagem advoguem a favor do reuso de todos os subprodutos de projetos de *software*, na prática, o reuso do código-fonte é a mais utilizada. Segundo BIGGERSTAFF e RICHTER, exemplos deste tipo de reuso são os esqueletos de código, funções, programas e objetos Smaltalk e mecanismos de *pipe* do Unix (BIGGERSTAFF, 1987).

Para a utilização desta abordagem, é necessário selecionar e recuperar, entender, adaptar e integrar o componente. Os sistemas de bibliotecas são ferramentas típicas para a seleção e recuperação de componentes; a análise de domínio fornece técnicas para o entendimento e a adaptação; e as tecnologias de interfaces cuidam da integração.

Para que esta abordagem seja mais eficiente, é necessário dar foco na arquitetura de sistemas. Sem a definição de uma arquitetura, torna-se muito difícil criar componentes que possam ser empregados em várias aplicações. Muitos casos de fracasso são atribuídos a problemas de arquitetura de sistemas, como relata GARLAN no seu artigo (GARLAN, 1995).

2.5.9 - REUSO BASEADO EM GERADORES DE CÓDIGO

Esta é uma abordagem mais sofisticada do reuso de software e consiste no reuso no nível de especificação, através do emprego de geradores de código ou de aplicações. Apresenta maior retorno e maior potencial, porém, é uma tecnologia complexa e imatura para ser empregada na indústria de software (PRIETO-DÍAZ, 1993).

Pesquisas sobre este tipo de reuso estão focadas em como formalizar a representação do domínio e dos processos de software e meta geradores. A análise de domínio têm sido um elemento importante para gerar vocabulários, arquiteturas e gramáticas para um domínio específico. A tendência caminha rumo aos meta-geradores com aplicação de um domínio específico de geradores onde o reuso seria naturalmente desenvolvido.

2.5.10 - REUSO CAIXA-PRETA

O reuso do tipo caixa preta é aquele onde não ocorre modificações no componente. Tipicamente, os componentes são encapsulados e as suas funcionalidades acessadas através de interfaces padronizadas. Embora esta abordagem garanta maior qualidade e confiabilidade, o seu custo é maior do que o caso anterior para a criação de componentes (PRIETO-DÍAZ, 1993).

Por outro lado, esta abordagem aproxima a construção de um sistema software do um sistema de hardware. Pode-se encontrar na literatura, algumas abordagens que tratam o reuso de software de maneira similar ao reuso de hardware.

O grande avanço atual do reuso está na abordagem de reuso do tipo caixa-preta. O uso de componentes de software impulsionam este tipo de reuso e os custos de desenvolvimento de componentes específicos são compensados pela possibilidade de aquisição de componentes mais genéricos no mercado. Com isso, esta abordagem se mostra como um dos meios mais promissores de se conseguir um sistema confiável num tempo menor.

2.5.11 - REUSO CAIXA BRANCA

O reuso do tipo caixa branca prevê a modificação e a adaptação dos componentes. É um tipo bem utilizado, especialmente quando o modo de reuso ad-hoc. A maioria dos processos de reuso, bem como as fábricas de software, utilizam este tipo (PRIETO-DÍAZ, 1993).

A principal dificuldade neste tipo de reuso é o controle das mudanças dos componentes, que precisam ser mais formalizados. Para evitar as constantes mudanças, muitas empresas estão parametrizando os seus reutilizáveis para torná-los mais flexíveis, porém aumentando a complexidade dos mesmos e perdendo a eficiência da execução da aplicação.

2.5.12 - REUSO DE CÓDIGOS FONTE

O tipo de reuso mais utilizado na prática é o reuso de código. A maioria das ferramentas de reuso, ambientes e métodos é voltada para o reuso de código. Outros aspectos do reuso estão sendo enfocados, e a atenção dada aos códigos-fonte está diminuindo. Eventualmente o uso dos geradores automáticos de código pode aumentar, fazendo com que os desenvolvedores não precisem mais lidar com código-fonte, da mesma maneira que não precisam mais lidar com a linguagem Assembly.

2.5.13 - REUSO DE PROJETOS

Reuso de Projetos oferecem um retorno maior do que o reuso de código. Quanto maior o nível do componente, maior ganho se obtém, dado que os subprodutos gerados também serão componentes. Neste sentido, ao se reusar projetos, o reuso de código ou de módulos executáveis é uma conseqüência direta.

Nesta abordagem, as especificações são utilizadas para identificar as partes de projeto que podem ser reusadas. A análise de domínio também é essencial para que se possa reusar um projeto. Atualmente, o reuso de projetos é realizada com bastante freqüência por meio da utilização de técnicas orientadas a objetos. Os vários trabalhos sobre padrões mostram a sua praticidade (GAMMA, 1995)

2.5.14 - REUSO DE ESPEFICIFICAÇÕES

Da mesma maneira que ocorre com projetos, quando se reutiliza uma especificação, tem-se, como conseqüência direta o reuso de projeto e do código-fonte.

O reuso de especificação apresenta o maior retorno e é o objeto de estudo do reuso de gerador de códigos (GAMMA, 1995).

2.5.15 - REUSO DE OBJETOS

Há muitos métodos, ferramentas, e ambiente para criar os objetos. As técnicas orientadas a objetos buscam a integração de ferramentas e métodos para cobrir o desenvolvimento de todas as fases. A técnica orientada a objetos é vista como a técnica do futuro para o reuso. Pode dar suporte ao reuso planejado, vertical e composicional (GAMMA, 1995).

2.5.16 - REUSO DE TEXTOS

Como todos subprodutos são destinadas às pessoas, com exceção do código-fonte, a importância do reuso do texto é bastante grande. Apesar de estar ainda em fase de pesquisa, seu conhecimento e técnicas disponíveis já prometem um progresso rápido. Hipertextos também são chaves para a tecnologia, a tendência encaminha à integração dos textos reusáveis com todos os subprodutos (GAMMA, 1995).

2.6 - REPOSITÓRIO DE DADOS PARA REUSO

Webster's Dictionary define a palavra repositório como "qualquer coisa ou pessoa considerada como um centro de acúmulo ou armazenagem" (WEBSTER, 1974).

Durante os primórdios do desenvolvimento de software, o repositório era, na verdade uma pessoa – o programador que tinha de se lembrar da localização de todas as informações pertinentes a um projeto de software, que tinha de se lembrar de informações que nunca foram escritas e reconstruir informações que se haviam perdido. Infelizmente, usar uma pessoa como "o centro de acúmulo e armazenagem" (ainda que isso seja aderente à definição do Webster) não é muito adequada. Hoje, o repositório é um arcabouço – um banco de dados que atua como o centro de tanto de acúmulo como de armazenagem das informações de engenharia de software. O papel da pessoa (engenheiro de software) é interagir com o repositório, usando ferramentas CASE que estejam integradas ao repositório (PRESSMAN, 1995).

Uma série de diferentes termos é usada para se referir ao local de armazenagem para as informações de engenharia de software: banco de dados CASE, banco de dados de projeto, banco de dados de ambientes integrados de suporte a projetos e repositório. Ainda que existam sutis diferenças entre alguns desses termos, todos se referem à um repositório que é imaginada como o centro de acúmulo e armazenagem (PRESSMAN, 1995).

O repositório de dados apresenta funções básicas de um sistema de gerenciamento de banco de dados, mas, além disso, o repositório realiza as seguintes funções (FORTE, 1989):

Integridade de dados - Inclui uma função para validar as entradas no repositório, garante a consistência entre objetos relacionados e executa automaticamente modificações quando uma mudança em um objeto exige mudanças em objetos que estejam relacionados a ele.

Compartilhamento de informações - Oferece um mecanismo para o compartilhamento de informações entre múltiplos desenvolvedores e entre múltiplas ferramentas, gerencia e controla acesso a dados de multiusuários e bloqueia/desbloqueia objetos de forma que as mudanças não sejam inadvertidamente sobrepostas umas às outras.

Integração dados – Estabelece um modelo de dados que pode ser disponível a todas as ferramentas, controlam o acesso aos dados e realiza funções de gerenciamento de configuração apropriadas. Estabelece o sistema de gerenciamento de banco de dados que relaciona objetos de dados de forma que outras funções possam ser realizadas.

Imposição metodológica - Define um paradigma específico para a engenharia de software que é implícito no modelo de dados armazenado no repositório; no mínimo, os relacionamentos e objetos definem um conjunto de etapas que deve ser cumprido para construir o conteúdo do repositório.

Padronização de documentos - Leva diretamente a uma abordagem padrão para a criação de documentos de engenharia de software ao criar definições para os objetos do banco de dados.

Para realizar essas funções, o repositório é definido em termos de um metamodelo. O metamodelo determina como as informações são armazenadas no repositório, como os dados podem ser obtidos pelas ferramentas e visto pelos engenheiros de software, quão bem a segurança e a integridade dos dados podem ser mantidas e quão facilmente o modelo existente pode ser ampliado para atender as novas necessidades. O

metamodelo é o gabarito no qual as informações de engenharia de software são colocadas (WELKE, 1989).

3 - ENGENHARIA DE DOMÍNIO

A Engenharia de domínio é o processo de identificar e organizar o conhecimento sobre uma classe de problemas (o domínio do problema) para suportar sua descrição (PRIETO-DIAZ, 1991). Segundo PRESSMAN (2001), os objetivos da Engenharia de Domínio são identificar, construir, catalogar e disseminar um conjunto de componentes que tenha aplicabilidade em softwares já existentes e novos.

Com o reuso sistemático, torna-se necessário criar processos de engenharia de domínio como parte do processo de desenvolvimento, independente da Engenharia de *Software*. A Figura 1 ilustra os processos da Engenharia e Desenvolvimento Baseado em Componentes.

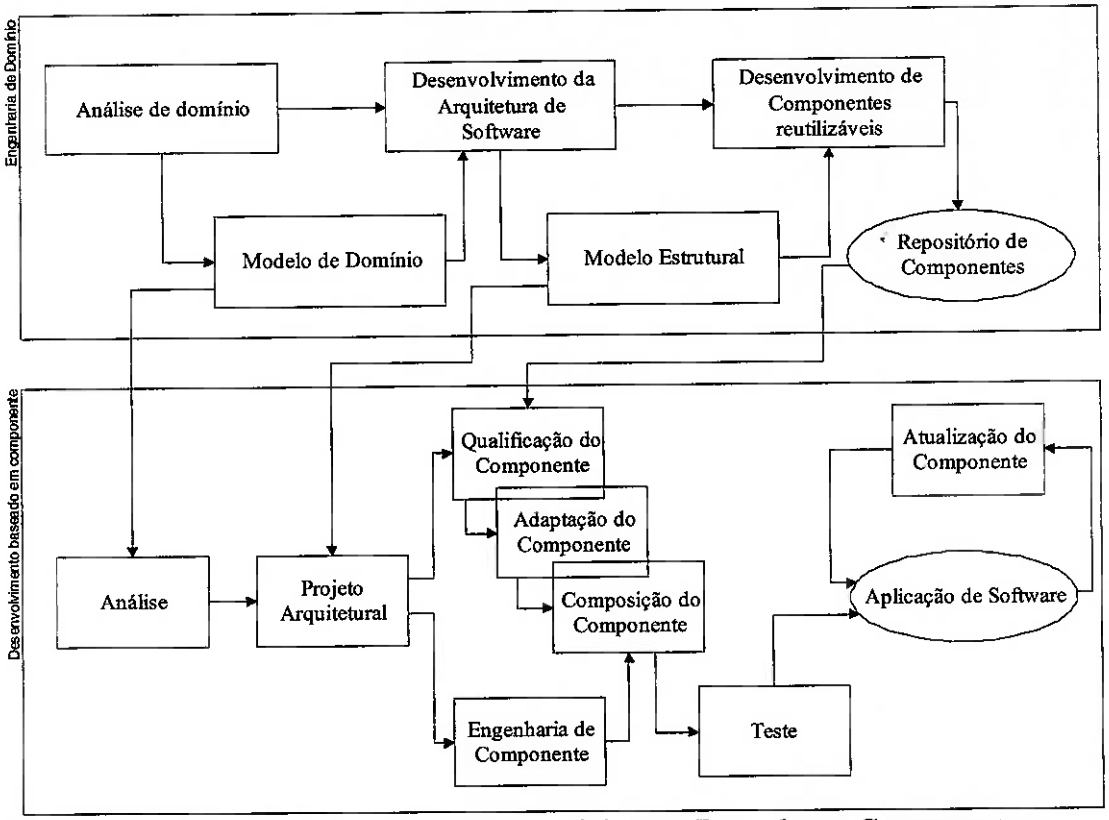

Figura 1 - Engenharia de Domínio e Desenvolvimento Baseado em Componentes (PRESSMAN, 2001).

O modelo de processo de Engenharia de Domínio e Desenvolvimento Baseado em Componentes (Figura 1) enfatiza trilhas paralelas, onde a Engenharia de Domínio ocorre concorrentemente à atividade de Desenvolvimento Baseado em Componentes. A Engenharia de Domínio executa o trabalho requerido para estabelecer um conjunto de componentes que possam ser reutilizados pelo engenheiro de software durante o Desenvolvimento Baseado em Componentes.

Como ilustrado na Figura 1, a atividade de Engenharia de Domínio inclui três grandes fases: análise de domínio, projeto de domínio (desenvolvimento da arquitetura de software), implementação do domínio (desenvolvimento de componente reutilizável) seguidas de modelo de domínio, modelo estrutural e repositório de componentes. Na análise de domínio são definidos os principais conceitos do domínio ressaltando suas similaridades e diferenças em um alto nível de abstração. O projeto de domínio detalha os conceitos e as características importantes no domínio, construindo arquiteturas de software, a fim de atender os requisitos identificados no modelo de domínio. A implementação do domínio consiste na codificação dos componentes para compor a arquiteturas de software, com ênfase na definição precisa de suas interfaces, armazenando-os em um repositório de componentes.

A atividade de Desenvolvimento Baseado em Componentes ocorre paralelamente à da Engenharia de domínio. Utilizando métodos de análise e projeto arquitetural, a equipe de software refina um estilo arquitetural apropriado para o modelo de análise criado. Uma vez definida a arquitetura, deve-se preenchê-la com componentes disponíveis nos repositórios de reuso ou criados para atender às necessidades requeridas.

Segundo PRESSMAN (2001), quando se propõe reusar componentes, sua existência não garante que sua integração em uma determinada arquitetura seja realizada de forma fácil e eficiente, pois alguns destes componentes reusáveis são desenvolvidos *in-house* ou extraídos de aplicações existentes, e até mesmo adquiridos de terceiros.

Por esta razão, uma seqüência de atividades do Desenvolvimento Baseado em Componentes deve ser aplicada, tais como, (1) qualificar: garante que componentes candidatos executem a função requerida, ajustando-se corretamente ao estilo arquitetural e exibindo as características de qualidade que são requeridas para a aplicação; (2) adaptar: componentes reutilizáveis que não combinam com as regras do projeto arquitetural devem ser adaptados para corresponder às necessidades requeridas ou descartados e substituídos por outros mais apropriados; e (3) compor: integra os componentes qualificados, adaptados e construídos em uma arquitetura estabelecida que provê uma infra-estrutura capaz de ocultar tais componentes.

Resumidamente, o Desenvolvimento Baseado em Componentes se preocupa com a criação de componentes que possam ser reutilizados em outras aplicações. Para que a reutilização possa ser efetiva, deve-se considerá-la em todas as fases do processo de desenvolvimento do software.

3.1 - ANÁLISE DE DOMÍNIOS

A análise de domínio foi introduzida por J.M. Neighbors (1981) como atividade para identificar objetos e operações de uma classe de sistemas similares em um problema de domínio. A análise de domínio é a chave para o reuso de software sistemático, formal e efetivo. E através da análise de domínio que o conhecimento é transformado em especificações genéricas, projetos e arquiteturas. Formulários genéricos são a base para criar componentes fáceis de utilizar. Muitos métodos de análise de domínio tem sido proposto nos últimos anos, bem como ferramentas para o suporte do mesmo (PRIETO-DÍAZ, 1991).

Há várias maneiras de definir um domínio. Este termo geralmente é usado para definir um grupo ou conjunto de sistemas ou áreas funcionais, dentro de um sistema, que possuem funcionalidade semelhante (SEI, 1997). Berard (1992) criou ainda duas definições: "uma coleção de aplicações de software presentes ou futuras que compartilham um conjunto de características comuns" ou "um conjunto bem definido de características que descrevem com exatidão, com escopo definido, e

completamente, uma família de problemas para a qual soluções de aplicações de computadores estão sendo e serão encontradas" (BERARD, 1992).

As informações providas pelo domínio são coletadas de sistemas existentes na forma de código-fonte, documentação, projetos, manual de usuários e planos de teste, juntamente com conhecimento de domínio e requisitos para sistemas atuais e futuros. Desenvolvedores extraem informações e conhecimento relevantes, analisam e resumem dados do domínio, o conhecimento e o resumo são encapsulados em forma de modelos de domínio, padrões e coleções de componentes para reuso.

Todo o processo é conduzido por métodos de análise de domínio e por procedimentos gerenciais. Os modelos de domínio, avaliações do conteúdo e avaliações do processo de desempenho de reuso constituem o restante da infra-estrutura (PRIETO-DIAZ, 1993).

3.2 - PROCESSO DE ANÁLISE DO DOMÍNIO

Como enfatizado por Neighbors (1981), a chave do desenvolvimento de software com reuso de software é capturada pela análise de domínio com seu foco abrangente de promover o reuso de itens mais abstratos do que somente código. Entre estes itens podem ser citados como exemplos arquiteturas de software, soluções de projeto, modelos de requisitos e, idealmente, conhecimento. No entanto, essa atividade se mostrou complexa e geralmente cara, dependendo da criação de toda uma cultura de apoio no ambiente em que o desenvolvimento para reuso e com reuso vá ocorrer. Além disso, o sucesso depende diretamente do modelo gerado no que diz respeito à relevância e à qualidade da representação dos itens de informação, bem como de suas relações.

ARANGO (1989), salienta a dificuldade de se identificar, capturar e organizar estes itens apropriados de informação, de forma que se torna clara a necessidade de um processo bem definido e estruturado. Deste modo, a análise de domínio toma a forma de um conjunto interdisciplinar que envolve adaptações de técnicas comumente usadas nas áreas de engenharia de software, engenharia de requisitos, modelagem conceitual,

aquisição de conhecimento e representação de conhecimento, podendo ser vista como versão em meta-nível da engenharia de requisitos e como um processo de engenharia do conhecimento direcionado a suportar uma tarefa particular de resolução de problemas.

Segundo FIRESMITH (1993) a análise de domínio de software é redigida com a identificação, análise e especificação, de requisitos em comum a partir de uma aplicação específica de domínio, tipicamente para reuso em múltiplos projetos dentro da aplicação do domínio. A análise de domínio, por exemplo é a identificação, análise, especificação em comum e capacidade de reuso dentro de uma aplicação específica de domínio em termos de objetos em comum e classes.
A aplicação de análise de domínio tem como objetivo encontrar ou criar classes amplamente aplicáveis, que possam ser reutilizáveis.

A figura 2 (ARANGO, 1989) mostra que os recursos de conhecimento do domínio são reconhecidos e tentam identificar objetos que possam ser reusáveis. O engenheiro de instrução investiga a área específica de interesse, na tentativa de extrair os fatos que possam ser usados na criação de um sistema técnico.

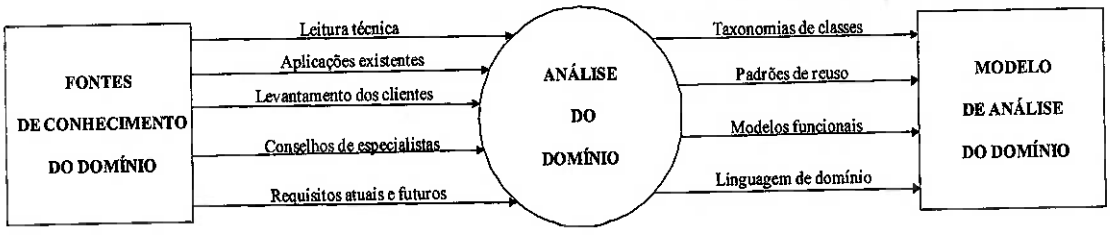

Figura 2 - Entrada e saída da Análise do domínio (ARANGO, 1989).

O processo de análise de domínio pode ser caracterizado por uma série de atividades, começando com a identificação do domínio, que será investigado, e finalizando com a especificação dos objetos e classes que caracterizam o domínio. Berard (BERARD, 1993) sugere as seguintes atividades:

- Definir o domínio a ser investigado.
- Categorizar os itens extraídos do domínio.
- Coletar uma amostra representativa das aplicações no domínio.
- Analisar cada amostra.
- Desenvolver um modelo de análise para objetos.

Complementando estes passos, o analista de domínio deverá criar um conjunto de guias de reuso e desenvolver um exemplo, ilustrando como os objetos de domínio poderão ser utilizados para criar uma nova aplicação.

Quando o negócio, sistema ou o produto de um domínio são definidos como a estratégia do negócio, um esforço contínuo em criar uma biblioteca de reuso pode ser experimentada. O objetivo é ter habilidade de criar um software com base no domínio que apresente uma alta porcentagem de componentes reusáveis. Baixo custo, alta qualidade e melhoria do tempo são argumentos a favor dos esforços dedicados pela análise de domínio (PRESSMAN, 1995).

Dentre as várias metodologias de Análise de Domínio existentes, Succi (2000) fez uma breve descrição conforme a tabela 2, das principais características de algumas delas.

Metodologia	Características
Produto – Paradigma Orientado [McCain 85]	A caracterização do domínio (chamada de "análise do mercado") é descrita. Não são criados modelos , e sim um conjunto de entidades para o reuso, não-entidades e abstrações.
Análise de domínio para reusabilidade [Prieto-Diaz 87]	O processo propõe atividades de classificação de bottom-up e atividades de análise de sistema top-dowm. O produto é uma coleção de componentes para reuso organizados e classificados.
FODA – Feature Oriented Domain Analysis Kang 90]	Focada na extração de características – aspectos visíveis – de um sistema, classificando-os em funcional, operacional e de apresentação. Usa técnicas estáveis, como diagramas ER ou OO, enfatizando uma visão funcional das aplicações.
Análise de domínio na síntese do meio ambiente [Campbell 90]	Metodologia similar à FODA. Tem uma fase de preparação dividida em descrição do domínio e qualificação do domínio. A qualificação foca em viabilidade e custos.
RAPID Análise de domínio [Vitaletti 90]	Metodologia (para linguagem ADA) que busca facilitar a reusabilidade. Os dados coletados de projetos de sistemas existentes ou com experts são guardados em um Banco de Dados de Informação de Domínio.
Idea Análise de domínio [Lubars 91]	Objetiva o reuso projetos de software abstratos, com operadores e passos para solucionar problemas. É cíclico e próximo à natureza evolucionária da Análise de Domínio
PROTEUS Análise de domínio metodologia [HP 94]	Identifica as partes comuns e não variantes de um sistema em um Domínio. A ênfase é o suporte ao desenvolvimento evolucionário. Usa técnicas de orientação a objetos e prevê a documentação explícita e a validação dos Modelos de domínio.

Tabela 2 - Principais características de metodologias de Análise de Domínio existentes (SUCCI, 2000).

3.3 - ARQUITETURAS DE DOMÍNIOS

Arquiteturas de domínios representam a estrutura global do projeto de um sistema de software. São estruturas de grande escala que representam a totalidade do sistema e são reusadas como um todo. Arquiteturas de domínio se assemelham a esquemas de software, no entanto, o foco principal são os subsistemas e as suas interações ao invés das estruturas de dados e algoritmos. As arquiteturas de domínio são bastante úteis quando se deseja representar o conhecimento acerca do domínio da aplicação e reutilizar este conhecimento em outras aplicações. Seus componentes apresentam uma pequena distancia cognitiva, pois é a própria arquitetura do domínio. O alto nível de abstração das arquiteturas tem a vantagem de não sofrer tanto com a rápida evolução dos domínios de implementação, pois as representações reforçam mais o domínio da aplicação e não de implementação, dando assim maior vida útil aos repositórios de componentes. (SHAW, 1989).

3.4 - ABORDAGEM DRACO

A abordagem DRACO (J.M. Neighbors, 1984 e 1989) é baseada na tecnologia de arquiteturas de domínios, onde cada arquitetura é encapsulada com seu próprio gerador de aplicações. Cada arquitetura de domínio tem sua linguagem própria do domínio e um conjunto de componentes que a implementa. A linguagem do domínio corresponde à especificação da abstração para uma determinada arquitetura, ela captura as abstrações relevantes (objetos e as operações) de um domínio particular.

Para a elaboração da arquitetura de um determinado sistema, Neighbors (1984) sugere os seguintes passos:

- Analisar sistemas semelhantes para o mesmo domínio de aplicação;
- Montar um modelo que seja a união das características dos sistemas semelhantes analisados;
- Determina as restrições do modelo e suas possíveis variações;
- Apresentar o modelo para especialistas para aprovação;
- Especificar a arquitetura usando a linguagem do domino;

Os domínios de aplicação variam desde abrangências restritas como listas e pilhas, até domínios mais abrangentes como sistemas de controle industrial, automação bancária, controle de tráfego aéreo e compiladores. Na abordagem DRACO os domínios podem ser classificados em três grupos descritos abaixo, que representam diferentes níveis de abstração.

a) Domínios de aplicações: encapsulam o conhecimento em volta da aplicação. Se o exemplo de aplicação for um sistema de controle de tráfego aéreo, deverão ser representadas na linguagem do domínio as características deste sistema, tais como: altura permitida para as aeronaves, planos de decolagem e aterrissagem, procedimentos de emergência, prioridades de operações e etc.

b) Domínios de modelagem: são modelos utilizados nas aplicações de computação, como interfaces como operador, banco de dados e sistemas operacionais.

c) Domínios executáveis: são linguagens de programação que geram código executável.

3.5 - FODA – FEATURE-ORIENTED DOMAIN ANALYSIS

É um método de Análise de Domínio baseado na identificação dos aspectos relevantes ou distintos de uma classe de sistemas. Foi criada a partir do estudo profundo de outros métodos de análise.

Segundo a descrição publicada pelo SEI (SEI, 1997) a partir do estudo da bibliografia original (KANG, 1990), a metodologia FODA parte de dois conceitos: a abstração e o refinamento. A abstração é utilizada para criar produtos do domínio a partir das aplicações específicas do domínio, que generalizam as funcionalidades e projeto das mesmas. Isto é conseguido através da abstração dos aspectos que fazem uma aplicação diferente da outra. No método as aplicações do domínio devem ser abstraídas até o ponto em que não exista nenhuma diferença entre elas. As aplicações específicas são então desenvolvidas através do refinamento dos produtos do domínio.

O processo proposto pelo método FODA tem como produto principal um *framework* estruturado de modelos relacionados, que catalogam os resultados da Análise do Domínio. É dividido em três etapas:

a) Análise de contexto - tem como objetivo definir o escopo do domínio. São também analisados os relacionamentos entre o domínio e os elementos externos. Os resultados são documentados em um modelo de contexto (diagrama de blocos, diagrama de estrutura, diagrama de fluxo de dados, etc.).

b) Modelagem do domínio - assim que o escopo estiver definido, esta fase provê passos para analisar os pontos em comum e as diferenças presentes nas aplicações, e produz um conjunto de modelos do domínio. Consiste em três atividades principais:

- **Análise de aspectos (features):** a visão de um cliente ou um usuário final (ou seja, um especialista no domínio) sobre as capacidades e aspectos de uma classe de sistemas é capturada. Os aspectos descrevem o contexto do domínio de aplicações, as operações necessárias e seus atributos, e as variações de representação. São exemplos: descrição de funções, descrição dos objetivos e padrões de uso, requisitos de performance, exatidão, sincronização, etc. Podem ser definidos como alternativos, opcionais ou mandatários, sendo que estes últimos representam os aspectos básicos e seus relacionamentos. Os alternativos e opcionais representam as especializações que podem ocorrer (ou seja, que mudanças geralmente acontecem em situações diferentes). Para maior eficiência, o ideal é que os aspectos sejam modelados em uma ferramenta que possua uma linguagem baseada em regras, para que as dependências entre os aspectos possam ser mantidas e entendidas.

- **Análise de informações:** o conhecimento sobre o domínio (teorias científicas relevantes, práticas de engenharia, capacidades e uso de sistemas existentes, experiência passada de desenvolvimento e manutenção de produtos, decisões de projeto, histórico das mudanças de projeto, etc.) e os dados necessários para implementar aplicações são definidos e analisados. O objetivo desta fase é representar o conhecimento sobre o domínio em termos de entidades e relacionamentos entre estas, e disponibilizá-las para a derivação de objetos durante a Análise operacional e a Modelagem da Arquitetura. O modelo gerado pode ser um modelo Entidade Relacionamento (ER), uma rede semântica, ou um modelo Orientado a Objetos (OO).

- **Análise operacional:** é a fase onde são identificadas as características comportamentais (pontos comuns e diferenças em fluxos de dados ou fluxos de controle, modelos de máquina de estados, etc.) das aplicações do domínio. As funções comuns encontradas no domínio são estruturadas e seqüenciadas em um modelo operacional. O controle e o fluxo de dados de aplicações específicas podem ser instanciados ou derivados do modelo operacional através das adaptações apropriadas.

c) Modelagem da arquitetura - esta etapa cria uma solução de software para as aplicações do domínio, na forma de um modelo de arquitetura, que é um projeto de alto nível para as aplicações no domínio. O foco é na identificação de processos concorrentes e módulos comuns orientados ao domínio.

Para representar os modelos gerados pelo método FODA cada caso deve ser analisado. Já foram utilizadas com sucesso várias ferramentas que suportam modelos orientados a objetos, modelos de entidade relacionamento e redes semânticas (KANG, 1990).

4 - PATTERNS E FRAMEWORKS

Patterns e *Frameworks* são conceitos estreitamente relacionados com a orientação a objetos. *Patterns* têm vários usos no processo de desenvolvimento de software orientado a objetos, formam um vocabulário comum que permite uma melhor comunicação entre os desenvolvedores, uma documentação mais completa e uma melhor exploração das alternativas de projeto. Reduz a complexidade do sistema através da definição de abstrações que estão acima das classes e instâncias. Um bom conjunto de padrões aumenta muito a qualidade do programa.

Também constituem uma base de experiências reutilizáveis para a construção de software. Funcionam como peças na construção de projetos de software mais complexos; podem ser considerados como micro-arquiteturas que contribuem para arquitetura geral do sistema.

Frameworks consistem em uma arquitetura concreta para o reuso que provê estruturas genéricas para uma família de softwares. Um *framework* pode ser adaptado para atender várias necessidades dentro de um determinado domínio. Ele não constitui uma aplicação completa, visto que não tem toda a funcionalidade específica da aplicação. Esta pode ser construída, adicionando funcionalidades a um ou mais *frameworks*, através do mecanismo de heranças e de instanciações dos seus componentes.

Um *framework* difere das bibliotecas de programação pois determina quais objetos e métodos devem ser usados quando da ocorrência de eventos.

Ao contrário dos *frameworks,* os *patterns* permitem o reuso de micro-arquiteturas abstratas sem uma implementação concreta. Os *frameworks* são implementados em uma linguagem de programação, enquanto que os padrões constituem formas de usá-las. Em geral, os *patterns* são menos especializados que os *frameworks*, podendo ser aplicados em um número maior de aplicações. Eles ajudam na definição e na documentação de um *framework.* Normalmente, *frameworks* maduros englobam dezenas de *patterns.*

4.1 – PATTERNS

Quando especialistas trabalham na solução de um problema é raro criarem soluções totalmente novas, em geral buscam problemas similares aos que já tenham sido resolvidos e reusa a essência da solução na resolução do novo problema (BUSCHMANN, 1996).

Segundo Fowler (1997), um padrão é uma idéia que foi utilizada num contexto prático e que provavelmente será utilizada por outros. Desta forma, muitas experiências em criação de modelos para projetos e a repetição contínua de problemas apresentados levam à especificação dos padrões.

Os padrões de projeto contribuem para sucesso do reuso de projetos e arquiteturas. Também, ajuda-nos a escolher alternativas de projeto o qual faz um sistema ser próprio para o reuso e evita alternativas que comprometam o reuso. O padrão de projeto ainda pode melhorar a documentação e a manutenção dos sistemas já existentes (GAMMA, 1995).

Patterns documentam experiências de projetos e não são inventados ou criados artificialmente. Uma característica importante dos padrões é disponibilizar para muitos um conhecimento adquirido. Entretanto, apesar de determinarem a estrutura básica para a solução de um problema particular, eles não especificam uma solução detalhada. O esquema da solução é especificado pela descrição dos componentes, suas funções e relacionamentos e as formas de colaboração entre eles. (BUSCHMANN, 1996).

Segundo Fowler (1997) padrões conceituais, incluídos os padrões de projeto, são aqueles que representam a maneira como as pessoas pensam, mais do que a maneira como um sistema é planejado. O modelo conceitual é um privilégio humano de modo que o que buscamos é a sua aplicabilidade. Fowler (1997) descreve três partes essenciais que os padrões devem ter, ressaltando a forma como muitos autores vem descrevendo seus padrões. Esta forma é importante porque sustenta a definição de um padrão como a solução de um problema num contexto. As partes essenciais são:

- uma declaração do contexto onde o padrão é utilizado;
- o problema que ele busca solucionar;
- as forças que atuam na formação da solução.

Buschmann (1996) adota um esquema de três partes para documentar padrões: o contexto, o problema e a solução. O contexto estende a dicotomia problema-solução descrevendo a situação na qual o problema ocorre. Especificar um contexto, é uma tarefa difícil, pois é praticamente, impossível prever todas as situações onde um padrão pode ser utilizado. Uma solução apresentada é listar todas as situações conhecidas onde à utilização do padrão pode ser relevante.

Os padrões ajudam na redução da complexidade de muitas situações na vida real e podem fornecer combinações de ações que levam à solução de problemas (PREE, 1995). A idéia original provem do arquiteto Christopher Alexander (ALEXANDER, 1979) onde cada padrão descreve um problema que ocorre constantemente em um ambiente, e depois descreve seu núcleo (solução para aquele problema), de uma forma que a solução poderá ser usada milhões de vezes, sem a necessidade de realizar a tarefa duas vezes. Alexander propôs o uso de módulos padrões que fossem normalmente utilizados na maioria dos projetos, como itens básicos de um repositório de padrões (*patterns*), que seriam então combinados das mais diversas maneiras para compor um projeto arquitetônico (PREE, 1995).

Esta idéia foi estendida para o projeto e implementação de software. O conceito central está em identificar padrões conceituais nos projetos de software, ou seja, identificar os traços comuns em diferentes projetos que pudessem ser utilizados em outras soluções. A identificação destes traços comuns, sua explicitação e sistemática e chamada de *parttern.*

A abordagem de *patterns* é considerada como de alto potencial de reuso (GAMMA, 1995), pois um padrão de identificado e documentado pode ser reusado em diversas aplicações, como na idéia original de (ALEXANDER, 1979).

Em geral um *pattern* possui quatro elementos essenciais:

Nomear um *pattern*: serve para descrever o nome do projeto, suas soluções e conseqüências apenas em uma ou duas palavras. Ter um vocabulário para *patterns* ajuda-nos na nossa documentação e conversar sobre o assunto com os colegas. Encontrar bons nomes tem sido uma tarefa difícil para o desenvolvimento do nosso catalogo.

O problema: descreve quando aplicar um *pattern.* Descreve o problema e seu contexto. Pode descrever problemas específicos do projeto como qual a forma de representar algoritmos como objetos. Também pode descrever estruturas de classe ou objeto em um projeto flexível. Algumas vezes, haverá uma lista de condições no problema, a qual deverá ser encontrada antes de aplicar o *pattern.*

A solução: descreve os elementos que compõe o padrão, suas relações, responsabilidades e colaboração. A solução não descreve um projeto concreto ou a implementação de forma particular, porque o *pattern* é como um modelo que pode ser aplicado em várias situações diferentes. O padrão produz uma descrição abstrata do problema do projeto e este, é solucionado pelo arranjo geral de elementos.

Conseqüências: são os resultados da aplicação do padrão. Embora as conseqüências normalmente não sejam comentadas, quando descrevemos as decisões do projeto, são julgadas para avaliar as alternativas e entender o custo e benefício da aplicação do padrão.

Como elementos básicos de um *Pattern* Gamma (1995) sugere:

Nome - Devem expressar a essência do padrão. Um bom nome é vital, pois vai se tornar parte do vocabulário do projeto.

Propósito - O que faz o padrão de projeto? Qual o problema que se propõe a resolver? Qual os objetivos que deseja alcançar?

Motivação - Descreve o cenário no qual se aplica, todas as forças que estão presentes, as classes e os objetos relacionados.

Aplicabilidade - Quais são as situações na qual o *pattern* pode ser aplicado? Como reconhecer estas situações?

Participantes - Descreve as classes e ou objetos que participam no padrão de projeto e suas responsabilidades.

Colaborações - Descreve como os participantes colaboram entre si para realizar suas responsabilidades. Assim, a solução descreve não somente a estruturas estáticas, mas também a dinâmica comportamental dos objetos e classes.

Diagrama - Representação gráfica do padrão utilizando a técnica de modelagem de objetos.

Conseqüências - Quais os resultados obtidos com a aplicação padrão ? O que foi resolvido, o que não foi resolvido e que *patterns* podem ser aplicados neste momento?

Implementação - Quais são as recomendações e as técnicas que devem estar claras quando da implementação do padrão. Há questões relativas a uma linguagem específica.

Exemplos - Exemplos de sistemas reais, onde o padrão de projeto foi aplicado e que transformações ocorreram dentro de cada contexto.

Padrões de projeto afins – Quais padrões de projetos têm relação com este *pattern*? Quais são as diferenças importantes? Com que outros padrões este deve ser usado?

Os onze elementos descritos acima são a definição do *pattern.* Quanto mais claro e detalhado for o *pattern,* maiores possibilidades de reuso ele terá. A técnica de identificar e documentar os *patterns* pode ser utilizada em conjunto com métodos convencionais de desenvolvimento, aumentando o potencial de reuso dos componentes elaborados.

4.2 – FRAMEWORKS

Framework pode ser definido sob dois aspectos: estrutura e função. Quanto à estrutura, um *framework* é o reuso projeto de parte ou totalidade de um sistema representado por classes abstratas e pela forma como instâncias destas classes interagem. Em outras palavras, a estrutura de um *framework* consiste de um diagrama de classes e um diagrama de eventos (JOHNSON, 1997).

Quanto à função, um *framework* é o esqueleto de uma aplicação que pode ser personalizado por um desenvolvedor de aplicação. Em outras palavras, a função do *framework* é extrair e disponibilizar a essência de uma determinada família de aplicações, de modo que o desenvolvedor possa partir de um patamar superior de implementação da arquitetura. Neste sentido um *framework* tem a mesma função que uma API (Application Programming Interface). (JOHNSON, 1997).

Um *framework* pode ser descrito como uma arquitetura de software semidefinida que consiste de um conjunto de componentes individuais e de interconexões entre eles, de tal forma a criar uma infra-estrutura de apoio pré-fabricada para o desenvolvimento de aplicações de um ou mais domínios específicos. Portanto, um *framework* é mais do que uma hierarquia de classes é uma aplicação genérica que possui estruturas estáticas e dinâmicas, e pode ser reusada por muitas outras aplicações (LEWIS, 1995).

Um *framework* orientado a objeto é um conjunto de classes que são integradas e interagem de uma maneira bem definida para fornecer um conjunto de serviços que satisfaça uma solução total ou parcial para um problema. Dessa maneira, os *frameworks* têm se tornado popular, como um veículo para o reuso. Eles oferecem uma solução concreta para um problema de um determinado contexto. Utilizando a tecnologia de orientação a objetos, o desenvolvedor pode especializar, instanciar ou reusar as classes dadas, respeitando os relacionamentos já definidos (BARROCA, 1996).

Os *frameworks* devem necessariamente ser extensíveis a fim de permitir flexibilidade e abrangência nos domínios para os quais foram especificados, pois eles definem as arquiteturas sobre as quais várias aplicações serão desenvolvidas (OFFALI, 1996).

Uma vantagem derivada do reuso e extensão de funcionalidades, é que o processo de criação de uma aplicação torna-se mais fácil e demanda menos conhecimento específico do domínio da aplicação. Dessa maneira, o desenvolvedor necessita apenas saber como usar o *framework*, não requerendo conhecimento detalhado da complexidade abrangida por este. Assim, pouco código terá que ser projetado e implementado. Será necessário apenas o código adicional que corresponde às particularidades da solução procurada.

Lajoie (1994) baseia-se em uma escala gráfica que representa características de programação para desenvolver uma classificação de níveis de reuso e abstração em *frameworks*.

A Figura 3 apresenta os três níveis, nos quais observa-se que no reuso-em-baixa-escala são tratados elementos de baixo nível, como classes, métodos e ou fragmentos de código. No reuso-em-média-escala são envolvidas micro-arquiteturas isoladamente, assim como as interações entre micro-arquiteturas, aqui denominadas *frameworks*. E, no mais alto nível, o reuso-em-alta-escala, é considerado o reuso de aplicações, que são subsistemas independentes e podem ser reutilizados com poucas modificações ou extensões.

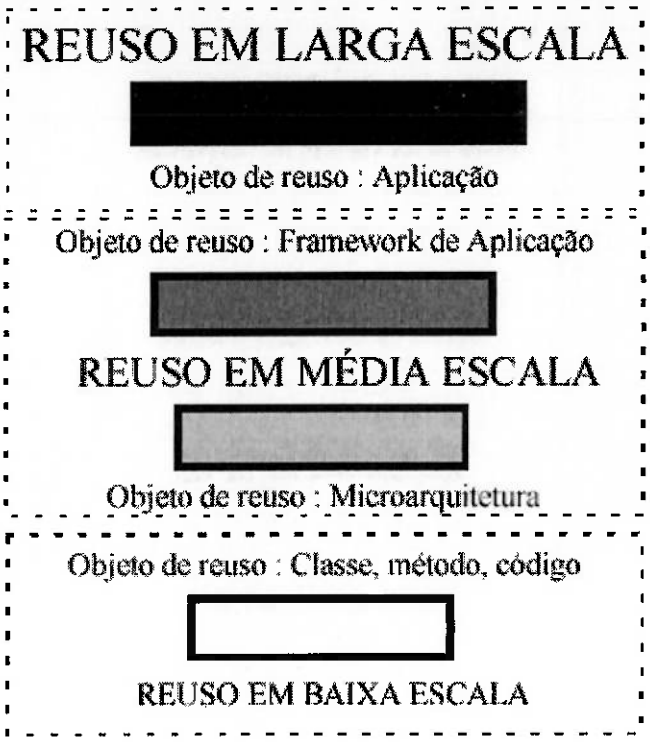

Figura 3 - Três níveis de reuso (LAJOIE, R , 1994).

Vindo ao encontro do reuso-em-larga-escala, Korson (1996), apresenta o axioma: "Os esforços de reuso limitados ao desenvolvimento e utilização de bibliotecas de classes não afetam fundamentalmente a produtividade do processo de desenvolvimento de *software*". Isto implica que o arcabouço para o reuso não seja composta somente por classes, mas contenha padrões, linguagens de *patterns*, *frameworks* e arquiteturas. Observa, ainda, que alguns desses recursos (ex. padroes e arquiteturas) são conceituais em natureza, ao passo que outros (classes e *frameworks*) são a implementação dos conceitos em código.

Korson (1996) afirma que classes como *string,* cliente, conta etc... elevam o nível de produtividade dentro de um paradigma, mas não alteram o paradigma de desenvolvimento de *software.* Para tal, o reuso em um nível mais elevado é requerido.

Quanto ao contexto de utilização do *framework* há uma classificação proposta em (FAYAD, 1997), segundo a qual os *frameworks* são classificados em: "*Framework* de Infra-estrutura de Sistema", "*Framework* de Integração de Middleware" e "*Framework* de Aplicação".

***Frameworks* de infra-estrutura** de sistema tratam de questões de projeto como sistemas operacionais, comunicação, interfaces gráficas e linguagens de programação. O *framework* AWT seria classificado como um *framework* de infra-estrutura de sistema.

***Frameworks* de integração de middleware** são responsáveis por integrar aplicações distribuídas e componentes em uma mesma arquitetura. Exemplos de *frameworks* de middleware são os ORB (Object Request Broker) como o CORBA - Common ORB Architecture, e o RMI - Remote Method Invocation.

Um *framework* de aplicação, freqüentemente referenciado na literatura sobre técnicas de reuso de software, consiste em uma aplicação própria para o reuso e semicompleta que pode ser especializada para produzir aplicações customizadas. Em contraste com as técnicas de reutilização de software mais antigas baseadas em bibliotecas de classes, os *frameworks* de aplicação estão direcionados para unidades de negócio particulares e domínios de aplicação específicos. *Frameworks* de aplicação tratam de questões de projeto de domínios de aplicação, como telecomunicações, finanças, produção e educação. (FAYAD, 1997).

5 - PROCESSOS DE DESENVOLVIMENTO COM REUSO DE SOFTWARE

Vimos anteriormente neste trabalho o princípio básico do reuso de software, no entanto, a efetivação desta idéia não se tem mostrado ser tão simples. Para que o reuso de software se torne uma realidade prática é necessária uma mudança na cultura das corporações, nos processos de desenvolvimento de software, nas ferramentas de suporte a estes processos e, naturalmente, é necessário que existam componentes em condições de serem reutilizados.

Para a efetivação do reuso de software, sejam eles pré-fabricados e pré-qualificados, é necessário que se tenha o processo de desenvolvimento orientado para o reuso de software. Durante o processo de desenvolvimento, a orientação será a de reusar ao máximo módulos desenvolvidos anteriormente. No caso de não existirem módulos na biblioteca que satisfaçam os requisitos desejados, novos módulos deverão ser desenvolvidos com base em métodos que facilitem o reuso futuro destes módulos.

O objetivo principal de introduzir reuso de software sistemático no processo de desenvolvimento é aumentar a produtividade dos grupos de desenvolvimento e melhorar a qualidade dos sistemas fornecidos. Os ganhos de produtividade serão alcançados a partir do momento em que se tenha uma biblioteca de módulos abrangente o suficiente para permitir um alto grau de reuso de software. A melhoria da qualidade virá pelo rígido controle sobre os módulos produzidos e armazenados na biblioteca, aumentando assim a chance do produto integrado ter a qualidade desejada.

Para que o objetivo do aumento da produtividade e melhoria da qualidade seja alcançado, o processo de desenvolvimento deve ser rigidamente gerenciado com pontos de controle e medição estrategicamente distribuídos ao longo do processo.

Outro fator muito importante para o processo de desenvolvimento com reuso de software, já abordado neste trabalho é a Análise de Domínio que faz parte da Engenharia de Domínio, que poderia ser definida, sinteticamente, como o processo de definição do escopo (definição do domínio), análise do domínio, especificação da estrutura (desenvolvimento da arquitetura do domínio) e implementação dos

componentes (requisitos, projeto, codificação e documentação) de uma classe de subsistemas que suportam reuso (KATZ, 1994).

Podemos dizer que a Análise de Domínio é o processo através do qual, a informação usada em sistemas em desenvolvimento de um domínio específico é, identificada, capturada, e organizada com a intenção de torná-la reutilizável em novos sistemas (PRIETO-DIAZ, 1990). Seu foco é o suporte para o reuso de software sistemático e em larga escala, o oposto de reuso de software oportunista (Ad-Hoc), que apresenta dificuldade de adaptação dos componentes a novos contextos. Isto é conseguido através da captura das características comuns e das variabilidades entre os sistemas do domínio, sendo assim não podemos deixar de salientar sua importância no processo de desenvolvimento.

Um dos maiores interesses na área de desenvolvimento de software atuais é provavelmente o vencimento do chamado paradoxo do software, que consiste na necessidade de produzir cada vez mais software, de maior qualidade, e em menos tempo. Sendo assim, isto explicaria o crescente interesse que vêm surgindo em torno da Análise de Domínio, voltada para a criação de infra-estrutura que permita o reuso de software de maneira eficiente. Prova deste interesse é a grande quantidade de metodologias existentes e o surgimento de várias novas na última década (AGRANGO, 1994).

Devido à sua importância no ambiente de software atual, onde vem crescendo cada vez mais a idéia da construção de componentes de software reutilizáveis, que possam ser amplamente distribuídos e aproveitados, o conhecimento e aplicação de uma ou mais metodologias de Análise de Domínio tende a tornar-se, também, uma necessidade dentro das organizações. Aquelas que tiverem a capacidade de reaproveitar o maior número de componentes reutilizáveis possível serão as que terão condições de vencer o paradoxo, e terão uma vantagem competitiva muito grande sobre as que não ultrapassarem esta barreira.

Abordamos também o *framework* que é um esqueleto formado por elementos abstratos de uma família de aplicações, passível de ser adaptado para atender necessidades de uma aplicação específica. No processo de desenvolvimento os *frameworks* e componentes de código são tecnologias complementares. *Frameworks* fornecem um contexto de reuso para componentes de código.

Como um *framework* é também um componente de código, é necessário que exista, anteriormente à sua implementação, uma atividade de projeto de software (projeto arquitetural e detalhado). Nessa fase, são realizados o projeto e a implementação do conjunto de características provindo da especificação de requisitos. Além das atividades comuns a fase de projeto no contexto da Engenharia de Software - são realizadas atividades como generalizações, especializações e recomposições que visem atingir uma estrutura mais estável.

No processo de desenvolvimento durante o Projeto e Implementação da infra-estrutura, o reuso de componentes também pode acontecer e em vários níveis. Primeiro, podem ser agrupados ao *framework* vários padrões que por ventura possam oferecer soluções para problemas de projeto. Também podem ser utilizados componentes de código durante a fase de implementação. Por fim, poderão existir casos em que um *framework* irá integrar outros *frameworks* menores ou mais específicos. Depois de implementado, o *framework* deverá ser classificado e armazenado em um repositório de componentes.

Discutem-se os esforços da engenharia de software na busca de estabelecer um processo de desenvolvimento com reuso de software maduro e bem estruturado. Desde a década de sessenta, quando da identificação da chamada crise de software, a comunidade de software vem buscando aplicar ao processo de desenvolvimento de software conceitos extremamente intuitivos às outras engenharias, como por exemplo, a idéia de sistemas abertos e interoperáveis e a prática de reutilização. A área mostrou uma constante evolução ao longo desses anos, com o advento de linguagens e paradigmas de desenvolvimento capazes de propiciar um nível mais adequado de abstração e com introdução de atividades de suporte como documentação, análise de risco, avaliação da qualidade e o próprio reuso de software.

Em termos genéricos pode-se identificar nos modelos de ciclo de vida conhecida a aplicabilidade potencial de conceitos de reúso discutidos neste trabalho.

A seguir, dois modelos de processos o modelo cascata (*waterfall*) e o modelo espiral (BOEHM, 1988) são analisados com a perspectiva de reúso de software.

5.1 - MODELO CASCATA (WATERFALL) E O REUSO DE SOFTWARE

O modelo clássico ou cascata, que também é conhecido por abordagem "top-town", foi idealizado em 1970 por Royce. Até meados da década de 1980 foi o único modelo com aceitação geral e tem como característica principal à seqüencialidade das atividades: sugere um tratamento ordenado e sistemático ao desenvolvimento do software. Cada etapa transcorre completamente e seus produtos são vistos como entrada para a nova fase; o software é desenvolvido em um longo processo e entregue ao final deste. Sugere-se laços de *feedback*, que permitem realimentar fases anteriores do processo, mas em geral o modelo cascata é considerado um modelo linear (ROYCE, 1970).

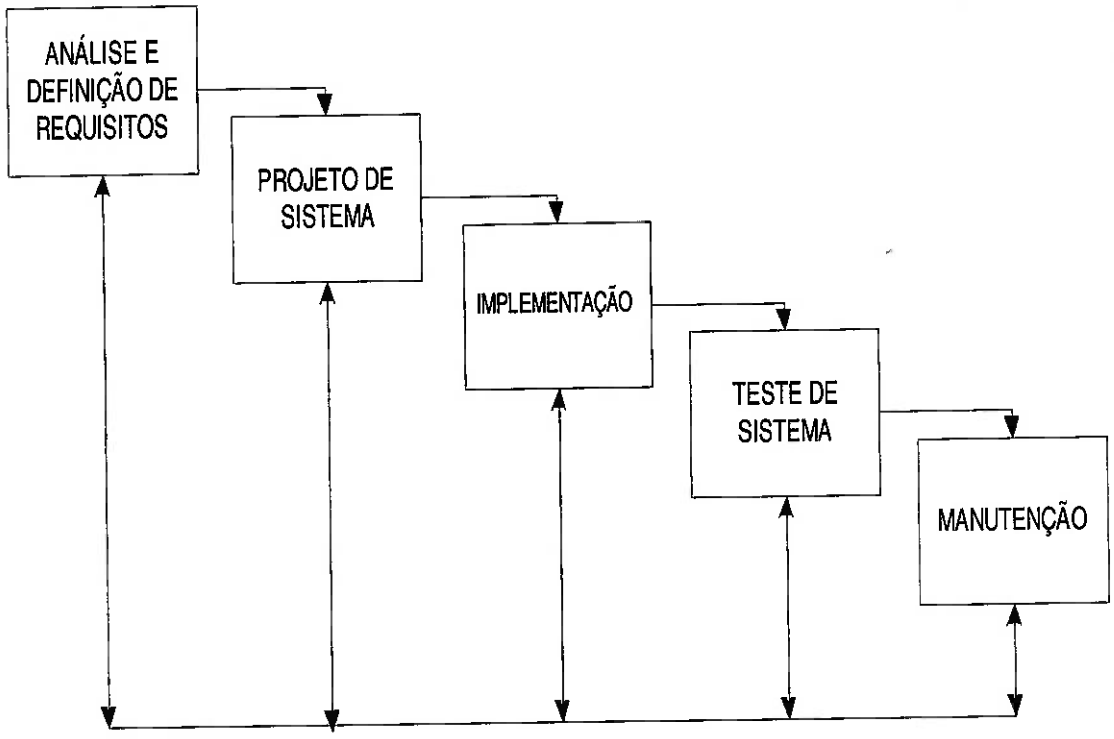

Figura 4 - Mostra o modelo Cascata (*waterfall*) (ROYCE, 1970).

O modelo Cascata é um modelo de engenharia projetado para ser aplicado no desenvolvimento do software. A idéia principal que o dirige é que as diferentes etapas de desenvolvimento seguem uma seqüência, a saída da primeira etapa flui para a segunda etapa e a saída da segunda etapa flui para a terceira e assim por diante. As

etapas são executadas seqüencialmente, de forma que uma etapa só poderá ter início quando a anterior tiver terminado.

5.1.1 - ANÁLISE E DEFINIÇÃO DE REQUISITOS

Nesta etapa, estabelecem-se os requisitos do produto que se deseja desenvolver, o que consiste usualmente nos serviços que se devem fornecer, limitações e objetivos do software. Sendo isso estabelecido, os requisitos devem ser definidos de uma maneira apropriada para que sejam úteis na etapa seguinte. Esta etapa inclui também a documentação e o estudo de viabilidade do projeto com o fim de determinar o início do processo de início de desenvolvimento do projeto do sistema; pode ser vista como uma concepção de um produto de software e também como o início do seu ciclo de vida.

Nesta primeira etapa do modelo a reutilização por meio de utilização de *patterns* pode facilitar e agilizar o ciclo, partes importantes como documentação e estudos de viabilidade podem ser comparados com os já existentes e utilizados com sucesso em outros projetos de forma otimizada. Podemos ressaltar nesta primeira etapa dois tipos reuso de software que podem auxiliar, o **reuso de procedimentos** e o **reuso de projetos** ambos com foco em utilização de conhecimentos em desenvolvimentos anteriores.

5.1.2 - PROJETO DE SISTEMA

O projeto do sistema é etapa de vários passos que se centraliza em quatro atributos diferentes do sistema: estrutura de dados, arquitetura do software, detalhes procedurais e caracterização das interfaces. A etapa de projeto representa os requisitos de uma forma que permita a codificação do produto. Da mesma maneira que a análise dos requisitos, o projeto é documentado e transforma-se em uma parte do software.

Como esta etapa gerará uma prévia etapa da codificação, é muito importante a utilização de componentes e patterns, qualquer tipo de reutilizável utilizado nesta etapa irá estruturar e agilizar a etapa seguinte que poderá utilizar muitos componentes. Nesta etapa podemos utilizar também vários tipos de reuso abordados neste trabalho. **Reuso de arquiteturas** relacionado com a arquitetura de software, **reuso de projeto** e **reuso de especificações**, pois nesta etapa de projeto irá ser especificado o requisito de uma forma que permita a codificação do produto.

5.1.3 - IMPLEMENTAÇÃO

Esta é a etapa em que são criados os programas. Se o projeto possui um nível de detalhe elevado, a etapa de codificação pode ser executada automaticamente. A princípio, sugere-se incluir um teste unitário dos módulos nesta etapa; nesse caso, as unidades de código produzidas são testadas individualmente antes de passar a etapa de integração e teste global.

Neste ciclo de implementação onde a ênfase maior está na codificação seguida de testes, podemos utilizar o reuso de software na sua forma mais elementar, o reuso de código. O reuso de código pode ser em muitos dos casos onde conseguimos a maior produtividade no reuso de software. Testes já utilizados anteriormente são também muito importantes nesta fase, pois podem certificar mais rapidamente os módulos gerados pelos componentes. **Reuso caixa preta** e também **reuso caixa branca** poderão ser aplicados nesta fase, porém vale lembrar que no caso do reuso caixa branca é muito importante aplicar um controle rígido das mudanças feitas para adaptação dos componentes ao novo projeto.

5.1.4 - TESTE DE SISTEMA

Concluída a codificação e os testes unitários, começa a fase de teste de sistema. A etapa de teste centraliza-se em dois pontos principais: as lógicas internas do software e as funcionalidades externas. Esta etapa decide se foram solucionados erros funcionais do software e assegura que as entradas definidas produzam resultados reais que coincidam com os requisitos especificados.

Como dito na etapa anterior, *patterns* de testes serão muito importantes nesta fase também, os *patterns* de testes dentro de um reuso sistemático torna-se, com o decorrer do tempo, cada vez mais eficazes, pois existe uma reciclagem aumentando a cada projeto o nível de maturidade dos componentes.

5.1.5 - MANUTENÇÃO

Esta etapa consiste na correção de erros que não foram previamente detectados. Esta etapa de manutenção situa-se à parte do ciclo de vida do produto de software e não pertence estritamente ao seu desenvolvimento.

Podemos ressaltar nesta etapa de manutenção, a revisão, modificação e melhoria dos componentes que voltarão para o repositório com mais qualidade para serem novamente reutilizados, este ciclo é importante, pois, assim podemos melhorar os componentes num ciclo de vida evolutivo.

O processo de desenvolvimento de um produto de software de acordo com o modelo Cascata é simples de conhecer e controlar. Passando pelas etapas do ciclo de vida do modelo cascata podemos identificar o potencial que o modelo possui para adotar o reuso de software em qualquer fase. Uma organização pode empregar a técnica de reuso de modo sistemático e deve haver procedimentos especiais para catalogar, recuperar e manter um conjunto de componentes para reuso.

Aplicando o reuso de software no modelo cascata podemos fornecer ao desenvolvedor resultados mais rápidos em cada etapa, pois o reuso pode orientar o desenvolvedor do software através de cada ciclo de vida, gerando diretrizes e suporte para passar por cada fase.

5.2 - MODELO ESPIRAL E O REUSO DE SOFTWARE

O Modelo Espiral foi originalmente proposto por Boehm (1988).
O objetivo do modelo espiral é prover um *metamodelo* que pode acomodar diversos processos específicos. Isto significa que podemos instanciar nele as principais características de outros modelos, adaptando-os às necessidades específicas de desenvolvedores ou às particularidades de um o software a ser desenvolvido. Este modelo prevê prototipação, desenvolvimento evolutivo e cíclico.

Sua principal inovação é guiar o processo de desenvolvimento gerado a partir deste metamodelo com base em análise de riscos e um planejamento que é realizado durante toda a evolução do desenvolvimento. Riscos são circunstâncias adversas que podem surgir durante o desenvolvimento de software comprometendo a evolução do processo ou diminuindo a qualidade do produto. São exemplos de riscos: pessoas que abandonam a equipe de desenvolvimento, ferramentas que não podem ser utilizadas, falhas em equipamentos usados no desenvolvimento ou que serão utilizados no produto final e etc.

Estaremos abordando o modelo espiral identificando em cada quadrante do modelo onde existe a possibilidade de reuso de software. É importante lembrar que uma vez iniciado o processo, mesmo que seja Ad-hoc em um determinado projeto, pode-se dar início a um processo de desenvolvimento voltado ao reuso efetivo e sistemático de software.

O modelo espiral descreve um fluxo de atividades cíclico e evolutivo constituído de quatro quadrantes. Vamos descrevê-los utilizando uma abordagem voltada ao reuso de software.

No **primeiro quadrante** devem ser determinados objetivos, soluções alternativas e restrições. Temos neste primeiro ciclo a possibilidade de implementação e utilização do reuso de software, a busca de soluções alternativas e restrições podem ser obtidas através de experiências de projetos anteriores ou mesmo em projeto em desenvolvimento, a busca de um *pattern* em um repositório especifico de reutilizáveis pode ajudar no ganho de tempo e qualidade. Podemos aplicar neste primeiro quadrante o **reuso de projetos**. No reuso de projetos a ênfase está no reuso parcial ou indireto de outros projetos é onde a prática sistemática do reuso de software está no nível de pesquisa.

No **segundo quadrante**, devem ser analisados os riscos das decisões do estágio anterior. Durante este estágio podem ser construídos protótipos ou realizarem simulações do software. Protótipos e componentes podem ser obtidos também em um repositório de componentes, porém a importância maior deste quadrante é porque podemos alimentar o repositório com novas e atualizadas informações.

O **terceiro quadrante** consiste nas atividades da fase de desenvolvimento, incluindo especificação, design, codificação e verificação. A principal característica é que a cada especificação que vai surgindo a cada quadrante - especificação de requisitos, do produto, da arquitetura, da interface de usuário e dos algoritmos e dados - deve ser feita à verificação apropriadamente. Neste quadrante os itens especificação, projeto e codificações também podem ser obtidos através de uma busca de componentes novamente com base em experiências e soluções (*patterns*) anteriores. Como esta fase consiste de especificação, design, codificação e verificação podemos utilizar vários tipos de reuso de software. O **reuso de idéias** para uma busca de soluções genéricas para uma classe de problema, **reuso de código**, **reuso de procedimentos** para auxiliar a fase de desenvolvimento, **reuso de composicional** por causa dos componentes de software e o **reuso baseado em geradores de código** que consiste basicamente na utilização de geradores de códigos e aplicações.

Por fim o **quarto quadrante** compreende a revisão das etapas anteriores e o planejamento da próxima fase. Neste planejamento, dependendo dos resultados obtidos nos estágios anteriores decisões, análise de riscos e verificação pode-se optar por seguir o desenvolvimento num modelo cascata (linear). Por exemplo, se já no primeiro quadrante, os requisitos forem completamente especificados e validados pode-se optar por seguir o modelo cascata. Caso contrário, pode-se optar pela construção de novos protótipos, incrementando-o, avaliando novos riscos e re-planejando o processo.

O modelo espiral completo está ilustrado na Figura 5. Cada setor da espiral corresponde a um estágio do desenvolvimento.

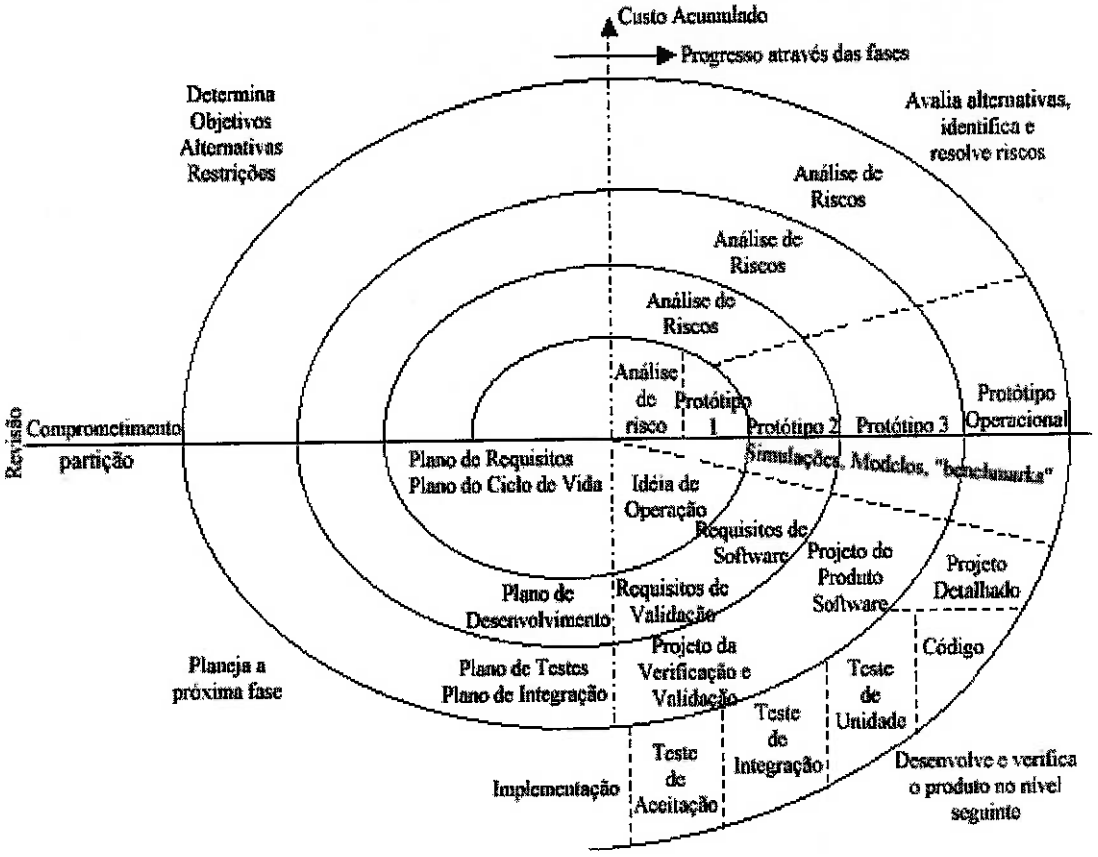

Figura 5 - Modelo Espiral de Boehm em 1988 (PRESSMAN, 1995)

O paradigma de modelo espiral é atualmente a abordagem mais realística para o desenvolvimento de softwares e sistemas em grande escala. Ele usa uma abordagem "evolucionária", capacitando o desenvolvedor e o cliente a entender e reagir aos riscos, em cada etapa evolutiva da espiral. Usa a prototipação como um mecanismo de redução de riscos, e a mesma pode ser aplicada em qualquer ponto evolutivo. Passando pelos ciclos desta abordagem podemos identificar como é importante a utilização do reuso neste modelo. Vimos que o processo de reuso de software pode ser aplicado na maioria dos quadrantes, e ainda renovando e reciclando o repositório de componentes.

Este capítulo discutiu a necessidade da integração sistemática da prática de reuso de software no processo de desenvolvimento, não temos um modelo especifico que podemos apresentar como o modelo ideal, porém pelo estudo do tema vimos que o reuso de software pode ser associado de maneira complementar ao desenvolvimento de software independente do modelo. Aplicamos a abordagem do reuso de software nos modelos cascata e espiral, e vimos que é possível sua utilização de maneira a facilitar o desenvolvimento de software, provendo qualidade e agilidade.

6 - CONCLUSÃO

Como mencionado anteriormente, este trabalho teve como objetivo principal abordar o reuso de software no processo de desenvolvimento, baseado em conceitos fundamentais, técnicos e gerenciais no processo de desenvolvimento. Abordamos a aplicação de técnicas de reuso de software com métodos convencionais de desenvolvimento, Engenharia de Domínio, Análise de Domínio, *Frameworks* e *Patterns*.

Como base teórica para se atingir este objetivo, foi feito um estudo das principais abordagens de reuso de software, seus métodos e técnicas, finalizando com uma análise de dois modelos de desenvolvimento (cascata e espiral), aplicando a abordagem de reuso de software em suas etapas. Identificamos com isso, que independente do modelo de desenvolvimento utilizado, o reuso de software pode ser inserido no contexto ajudando os desenvolvedores a entregar com mais rapidez e qualidade o produto do desenvolvimento.

Verificou-se também no estudo feito que existem alguns fatores que podem inibir reuso de software no processo de desenvolvimento. Fatores como o desconhecimento de técnicas e conceitos para a prática do reuso de software, falta de ferramentas especializadas e infra-estrutura.

Vimos também que além dos fatores técnicos e corporativos que podem inibir o reuso de software existe os fatores humanos, um bom exemplo disso são as gerências despreparadas para gerenciar projetos com reuso de software e a falta de hábito de trabalho em grupo, gerando quando muito bibliotecas pessoais.

Resumindo, o objetivo principal do reuso de software é o aumento da produtividade dos desenvolvedores. Porém, por trás da facilidade no entendimento do objetivo, existem uma série de fatores que tornam complexa a sua realização. Portanto, as principais contribuições deste trabalho centram-se na apresentação de uma visão panorâmica sobre o assunto, passando pelo levantamento das principais abordagens relacionadas ao tema até a aplicação prática de vários conceitos da área.

É possível com o auxílio deste trabalho que, gerentes, desenvolvedores e pesquisadores iniciantes no assunto de reuso de software, possam ter um maior esclarecimento dos conceitos básicos sobre o tema em questão. Os assuntos abordados neste trabalho poderão também servir de motivação para a continuidade de pesquisas sobre o tema.

6.1 - TRABALHOS FUTUROS

Ao final deste trabalho de pesquisa propõe-se como trabalhos futuros ampliar e melhorar os resultados deste trabalho. Dentre as possibilidades de extensões pode-se citar:

Sistemas com Maior Potencial de Reuso

- Sistemas que indicam ser um bom candidato a se aplicar técnicas de reuso de software.

Banco de Dados

- Estrutura de banco de dados para catalogação, classificação e recuperação de componentes.

Organização

Estrutura organizacional voltado ao desenvolvimento com reuso de software.

7 - REFERÊNCIAS BIBLIOGRÁFICAS.

ALEXANDER, C. **The Timeless Way of Building.** New York: Oxford University Press, 1979.

ARANGO,G.; PRIETO-DIAZ **Domain Analysis: Concepts and Research Directions**. Domain Analysis: Acquisition of Reusable Information for Software Construction, (Arango,G. And R. Prieto-Diaz, EDS). IEEE Computer Society Press, May 1989.

ARANGO,G. **Domain Analysis Methods** 17-49. Software Reusability. Chichester, England: Ellis Horwood, 1994.

BARROCA, L. H. **A framework and Patterns for the Specification of Reactive Systems**. In Proceedings of the EuroPLoP'96, July 1996.

BASILI, V. R.; BRIAND L.C. **Domain Analysis for the Reuse of Software Development Experiences**, 1996

BERARD, E. V. **Essays on Object-Oriented Software Engineering**, Addison Wesley, 1993

BERARD, E.: **Essays in Object-Oriented Software Engineering**. Prentice Hall, 1992.

BIGGERSTAFF, T.; RICHTER C. **Reusability framework, assesment, and directions**. IEEE Software, v.4, n. 2, p41-49, Mar., 1987.

BIGGERSTAFF,T.J.; A.J.PERLIS **Special Issue on Software Reusability**, IEEE Trans.Software Engineering, vol SE-10, Numero 5, Setembro de 1984.

BOEHM, B. **A Spiral Model for Software Development and Enhancement**. Computer, vol. 21, n. 5,pp. 61-72. may, 1988.

BURGALI, D. **The Framework Life Span, Communications of the ACM**, Vol. 40, Num. 10, Outubro. 1997.

BUSCHMANN, F. **A System of *Patterns***. John Wiley & Sons. 1996.

COBB,R.H. **In Praise of 4GLs, Datamation**, pag 92, 1985 Communications of the ACM, Vol. 40, Num. 10, Outubro. 1997.

DAVIS, T. **The reuse capability model: a basis for improving an organization's reuse capability**. International Workshop on Software Reusability in Herndon, VA, 1993.England: Ellis Horwood, 1994.

FAYAD, M. E. **Object-Oriented Application Frameworks**, CACM Vol. 40, No. 10, Outubro 1997.

FIRESMITH, D.G. **Object – Oriented Requirements Analysis And Logical Design**, Wiley, 1993

FORTE, G. **In Search of the Integrated Environment**, CASE Outlook, 1989.

FOWLER, M. **Analysis Patterns. Reusable Object Models**. Addison Wesley Longman, Inc, 1997.

FRAKES, W.; TERRY, C. **Software Reuse and Reusability Metrics and Models**, 1996.

FREEMAN, P. **Tutorial – Software Reusability**, IEEE Computer Society, 1987

GAMMA, E. et al. **Design *Patterns* – Elements of Reusable Object-Oriented Software**. Pg 1-31, 1995. Institute of Standards and Technology, 1994.

GARLAN, D.; ALLEN, R.; OCKERBLOOM, J. **Architectual Mismatch: Why Reuse is so Hard**. IEEE Software, v.12, n.6, p.17-26, Nov. 1995.

HUMPHREY, W. S. **Characterizing the Software Process: A Maturity Framework, IEEE Software,** March, 1988.

JACOBSON, I. et al. **Software Reuse: Architecture, Process and Organization for Business Success.** pg 80-169, 1a. ed., ACM Press, 1997.

JOHNSON, R. E. ***Frameworks*** **= (Components + Patterns). Communications of the ACM**, V. 40, 1997.

KANG, K.; COHEN, S.; HESS, J. **Feature-oriented domain analysis (FODA). Feasibility study. Software Engineering Institute, 1990.** http://www.sei.cmu.edu/str/descriptions/odm.html Acesso em 01 de nov. 2003.

KATZ, S., et al. **Glossary of Software Reuse Terms. Gaithersburg**, MD: National Institute of Standards and Technology, 1994.

KORSON, T. **Introduction to Reuse. Object Magazine**, v. 6, n. 1, April 1996.

LAJOIE, R. **Design and Reuse in Object-Oriented *Frameworks*: Patterns, Contracts, and Motifs in Concert**. In: 62nd Congress of the Association Canadienne Française pour Ávancement des Sciences (ACFAS), 1994, Montreal. Proceedings... Montreal - Canada, 1994.

LEWIS, T. et al. **Object-Oriented Application *Frameworks***, Manning publications Co, 1995.

McCLURE, C. **The Three Rs of Software Automation**. p. 221-63, Prentice Hall, 1992.

MCILROY, M. D. **Mass Produced Software Components, in Software Engineering**, P. Naur and B. Randell eds. NATO Science Committee Report, pp., Germany, p. 138-155, Oct., 1968.

NEIGHBORS, **JAMES M. Draco: A Method for Engineering Reusable Software Systems. In: Biggerstaff, Ted and Perlis, Alan (eds.): Software Reusability, 1989**

NEIGHBORS, **JAMES M. The DRACO Approach to Constructing Software From Reusable Components , 1984**

OFFALI, H. E. **The Essential Distributed Object Survival Guide** – Wiley Computer Publishing, 1996

PREE, W. **Design Patterns for Object-Oriented Software Development**. Addison Wesley. 1995.

PRESSMAN, R. S. Software **Engineering: A Practitioner's Approach**. 5. ed., New York, USA: McGraw Hill, 2001. pg 716 726

PRESSMAN, R. S. **Engenharia de Software**, São Paulo: Macron Books, Reusabilidade de Software. Pg 17, 101, 258, 318 . 1995

PRIETO-DIAZ, R.; ARANGO, G. **Domain analysis and software system modeling**. Califórnia: IEEE Computer Society Press Tutorial, 1991.

PRIETO-DIAZ, R: **Domain Analysis and Software Systems Modeling**. Los Alamitos, CA: IEEE Computer Society Press, 1991

PRIETO-DIAZ, R. **Reuse as a New Paradigm for Software-Development**. Proceedings of the International Workshop on Systematic Reuse, 1996

PRIETO-DIAZ, R: **Status Report: Software Reusability**. IEEE Software, v.10, n.3, p61-66, Maio 1993

ROTHENBERGER, M. A. **Systems Development with Systematic Software Reuse: an Empirical Analysis of Project Success Factors,** 1999