UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ADMINISTRAÇÃO – PPGA
PÓS GRADUAÇÃO EM REGULAÇÃO DE SERVIÇOS PÚBLICOS

Marco Aurélio Antunes

PROGRAMAÇÃO ORIENTADA A OBJETOS:
Estudo de Caso do Sistema de Ouvidoria Pública da AGERGS

Porto Alegre
2007

Marco Aurélio Antunes

PROGRAMAÇÃO ORIENTADA A OBJETOS:
Estudo de Caso do Sistema de Ouvidoria Pública da AGERGS

Trabalho de conclusão de curso de Especialização apresentado ao Programa de Pós-Graduação em Administração da Universidade Federal do Rio Grande do Sul, como requisito parcial para a obtenção do título de Especialista em Regulação de Serviços Públicos.

Orientador: Profa. Dra. Ângela Freitag Brodback

Porto Alegre
2007

Marco Aurélio Antunes

PROGRAMAÇÃO ORIENTADA A OBJETOS: Estudo de Caso do Sistema de Ouvidoria Pública da AGERGS

Trabalho de conclusão de curso de Especialização apresentado ao Programa de Pós-Graduação em Administração da Universidade Federal do Rio Grande do Sul, como requisito parcial para a obtenção do título de Especialista em Regulação de Serviços Públicos.

Conceito Final:

Aprovado em ________________________________

BANCA EXAMINADORA:

____

Prof. Dr. –

____

Prof. Dr. –

____

Prof. Dr. –

____

Orientador – Profa. Dra. Ângela Freitag Brodback – Ufrgs

AGRADECIMENTOS

Agradeço à AGERGS pela oportunidade de participar do Curso de Pós-Graduação em Regulação dos Serviços Públicos, à coordenação, aos professores e colaboradores do referido curso pelo incentivo e paciência, à orientadora Profa. Dra. Ângela Freitag Brodback por suas relevantes contribuições, aos colegas pela convivência, aos colegas do Núcleo de Informática pela dedicada solicitude, e por último mas não menos importante à minha família pela compreensão e estímulo.

RESUMO

O presente estudo aborda o paradigma de desenvolvimento de software da programação orientada a objeto. De forma sintética, mostra não apenas a evolução e arquitetura dos computadores, a evolução e as metodologias de programação de software, mas também a evolução e as características das linguagens de programação, e as estruturas e metodologias de desenvolvimento de software que abrange a Engenharia de Software. Faz um estudo de caso do SOA – Sistema de Ouvidoria da Agergs, software desenvolvido pelo Núcleo de Informática da Agergs, procurando investigar tanto as características da linguagem de programação utilizada, quanto o paradigma de desenvolvimento empregado nesse sistema. Os resultados mostraram que SOA foi desenvolvido com base na linguagem de programação Visual Basic, ao passo que esse software apresenta possibilidades de melhoria e atualização. A versão do Visual Basic adotada no desenvolvimento do SOA é dirigida por eventos e faz uso de objetos. No entanto, uma versão mais recente dessa linguagem é orientada a objeto e disponibiliza recursos para desenvolvimento de programas voltados para a internet. As melhorias apontadas pelos resultados sugerem a atualização do SOA para essa nova versão. Não é improvável que esse estudo possa servir de um modo geral como referência para o desenvolvimento de programas com orientação a objeto, e mais especificamente como suporte para novos projetos do Núcleo de Informática da Agergs.

ABSTRATC

The present work studies the object-oriented programming paradigm. Briefly, it shows the evolution and architecture of the computers, the evolution and the methodologies of software development, the evolution and the characteristics of the programming languages, and the structures and methodologies of the Software Engineering. It makes a case study of the software SOA (Sistema de Ouvidoria da Agergs) which was developed by the computer department of AGERGS (the agency of regulation from the State of Rio Grande do Sul). It investigates the programming language, and the development paradigm used in this system. The results had shown that the Visual BASIC was the programming language used to build SOA, and that this software presents some possibilities of improvement and update. The version of the Visual BASIC used is event-driven. However, a more recent version of this language is an object-oriented programming language. This study could be an initial reference for the object-oriented programming projects, and gives some support for new projects of the AGERGS' computer department.

LISTA DE ILUSTRAÇÕES

Figura 1: Genealogia das linguagens de programação 62
Figura 2: Genealogia do BASIC 66
Figura 3: IDE do Visual BASIC 6 87
Figura 4: Plataforma .NET 91
Figura 5: Formulário de Login..107
Figura 6: Formulário de Cadastro...107
Figura 7: Formulário de Pesquisa...108
Figura 8: Formulário de Pesquisa de Usuários...108
Figura 9: Formulário de Relatórios 109

LISTA DE ABREVIATURAS E SIGLAS

ADO – *ActiveX Data Objects*

AGERGS – Agência Estadual de Regulação dos Serviços Públicos Delegados do Rio Grande do Sul

ALGOL – *ALGOrithmic Language*

APT – *Automatically Programmed Tool*

ASP – *Active Server Pages*

BASIC – *Beginners All-purpose Symbolic Instruction Code*

CASE – *Computer-Aided Software Engineering*

CI – Circuitos Integrados

CLOS – *Common Lisp Object System*

CLR – *Common Language Runtime*

COBOL– Linguagem Orientada aos Negócios (*COmmon Business Oriented Language*)

COM *Component Object Model*

CPU – Unidade Central de Processamento (*Central Processing Unit*)

CPD – Centro de Processamento de Dados

CRC – Classe-Responsabilidade-Colaborador *(Class-Responsibility-Collaboration*)

DAO – *Data Access Objects*

DLL – Biblioteca de Ligação Dinâmica (*Dynamic-link library)*

DOS – Sistema Operacional (*Disk Operating System*)

FORTRAN – *FORmula TRANslator*

GPSS – *General Purpose Simulation System*

GUI – *Graphical User Interface*

HTML – *HyperText Markup Language*

IDE – *Integrated Development Environment*

LISP – *LISt Processor*

NIN – Núcleo de Informática

OLE – Vinculação e Incorporação de Objeto (*Object Linking and Embeldding)*

OMT – *Object Modeling Technique*

OOA – *Obect-Oriented Analysis*

OOD – *Object-Oriented Design*

OOAD– *Obect-Oriented Analysis and Design*

PROLOG – do francês: *PROgrammation en LOGique*

RAD – *Rapid Application Development*

RDO – *Remote Data Objects*

RPG – *Report Program Generator*

VB – Visual BASIC

VBA – *Visual BASIC for Applications*

WICS – Water Industry Commission For Scotland

SUMÁRIO

1 INTRODUÇÃO 11
2 TEMA 15
3 PROBLEMA 16
4 JUSTIFICATIVA 17
5 OBJETIVOS 18
5.1 OBJETIVO GERAL 18
5.2 OBJETIVOS ESPECÍFICOS 18
6 METODOLOGIA 19
7 REVISÃO DA LITERATURA 20
7.1 EVOLUÇÃO DOS COMPUTADORES 20
7.2 EVOLUÇÃO DO SOFTWARE 23
7.3 ARQUITETURA DO COMPUTADOR 25
7.4 DOMÍNIOS DA PROGRAMAÇÃO 26
7.5 CRITÉRIOS DE AVALIAÇÃO DA LINGUAGEM 27
7.6 METODOLOGIAS DE PROGRAMAÇÃO 28
7.7 LINGUAGENS DE PROGRAMAÇÃO 30
7.8 SISTEMAS BASEADOS EM COMPUTADOR 36
7.9 DOMÍNIOS DA INFORMAÇÃO 37
7.10 METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE 38
7.11 PROGRAMAÇÃO ORIENTADA A OBJETO 44
7.12 DESENVOLVIMENTO ORIENTADO A OBJETO 48
7.13 MÉTODO OOAD 50
7.14 CONCEITOS E PRINCÍPIOS FUNDAMENTAIS 57
7.15 LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETO 66
8 RESULTADOS 83
8.1 AGERGS 83
8.2 LINGUAGEM DE PROGRAMAÇÃO VISUAL BASIC 86
8.3 SISTEMA DE OUVIDORIA DA AGERGS 95
9 CONCLUSÃO 102

REFERÊNCIAS 103
ANEXO 108

1 INTRODUÇÃO

Os avanços tecnológicos em conjunto com o processo de globalização, condicionam a realidade brasileira dos anos 1990. Reduzidas taxas de desenvolvimento econômico e de altos desemprego e inflação, trazem a necessidade de reforma e reestruturação do Estado, nos seus aspectos administrativos e políticos e nas suas relações com a Sociedade e o Mercado. A tendência predominante em todo mundo é a dos Estados deixarem paulatinamente a produção de bens e serviços, delegando-as para o setor privado e passarem a focalizar a atenção na regulação econômica e na indução e coordenação do desenvolvimento. Surge, portanto, como resposta a esta grande crise econômica a superação de um Estado burocrático, produtor e empresário, por outro gerencial, regulador e promotor do desenvolvimento. Cabe ressaltar que não houve uma convergência dos países capitalistas avançados na forma de combinar liberalização e reforma da regulação econômica, o que se verificou foram diferenças marcantes nas soluções adotadas em cada país. (Pereira, 1997; Santana, 2002; Vogel, 1996 apud Fiani 1998)

A reestruturação dos setores de infra-estrutura no Brasil, como resultado da substituição do Estado pela iniciativa privada na sua operação, induz um processo de implantação de agências reguladoras que devem, fundamentalmente, ter a missão de incentivar e garantir os investimentos necessários, promover o bem-estar dos consumidores e usuários e aumentar a eficiência econômica. Em meados da década de 1990, como parte do plano de reforma do estado, começam a ser implantadas no país várias agências de regulação, as quais não são apenas um fenômeno nacional, mas surgem a partir dos anos 1980 em várias partes do mundo. O Estado do Rio Grande do Sul cria sua agência, a primeira no âmbito estadual, no ano de 1997, a AGERGS, uma autarquia com autonomia financeira, funcional e administrativa com objetivo de garantir a qualidade dos serviços públicos (energia, telecomunicações, transporte etc.) oferecidos aos usuários pelas concessionárias privadas e o equilíbrio econômico e financeiro dos contratos entre o poder

concedente (governo) e as empresas concessionárias. (Kann, 2002; Peci e Cavalcanti, 2000; Pires e Piccinini, 1999; Queiróz, 2002)

A criação de um órgão regulador dotado de autonomia é o pressuposto fundamental para assegurar credibilidade às reformas. A questão chave desta argumentação é a possibilidade das agências constituírem-se de instrumentos mais transparentes de regulação e controle de determinado setor da economia do que as antigas estruturas burocráticas. A estruturação da agência reguladora, portanto, deve garantir a sua eqüidistância em relação aos agentes sociais envolvidos, de maneira a dificultar a sua captura por qualquer área de interesse. Este modelo consiste em um triângulo, onde a agência reguladora situa-se no centro, o governo num dos vértices e os operadores (empresas concessionárias) e os usuários (consumidores dos serviços públicos) nos outros dois vértices, gerando uma eqüidistância entre a agência e os outros dois agentes. (Roxo, 2005)

Diferenças de objetivos não são raras entre governo e concessionárias prestadoras de serviços públicos. Por exemplo, o governo pode preocupar-se principalmente com novos investimentos, expansão dos serviços e baixos preços. Em contraste, o proprietário privado operador dos serviços procura a maximização de lucros. Objetivo que é geralmente entendido como sendo inconsistente com serviços amplamente disponíveis e preços baixos. Por causa dessa diferença de objetivos, os governos normalmente adotam instrumentos que levem as concessionárias a desenvolver atividades voltadas para atingir os objetivos governamentais. É consenso de que o desenvolvimento bem sucedido de serviços de infra-estrutura, tais como eletricidade, gás natural, telecomunicações, saneamento básico, etc., dependem em grande parte da adoção de apropriadas políticas públicas e de uma implementação eficiente das mesmas. O desenvolvimento de um aparato de regulação é um ponto crucial dessas políticas, proporcionando estabilidade, proteção dos usuários do abuso de poder de mercado, proteção de usuários e empresas do oportunismo político, bem como proporcionando incentivos aos fornecedores de serviços para que os mesmos operem eficientemente e realizem os investimentos necessários. (World Bank, 2004)

As estatísticas operacionais e os registros financeiros e contábeis da empresa concessionária, o comportamento da demanda, o nível de satisfação dos

usuários, em síntese, o conhecimento ou as informações estão distribuídos de forma desigual entre governo, concessionárias e usuários. Desigualdade geralmente conceituada como assimetria de informações, onde o operador privado possui acesso exclusivo e está de posse de informações sobre a empresa e o mercado. No caso das firmas operadoras de serviços públicos, cabe as agências de regulação exercerem essa importante função de superar a assimetria de informações existente a favor das empresas. Com o propósito de obter, analisar e divulgar as informações, as agências tipicamente podem fazer suas coletas em diversas fontes: nas concessionárias, no mercado financeiro, dos usuários, de um órgão governamental, de instituições privadas, etc. Tal atividade, vai depender dos objetivos, da capacidade e da estrutura das agências, além do custo e da dificuldade de obtenção dessas informações. (World Bank, 2004)

O conhecimento, portanto, é um dos pilares do processo regulatório. E, a informação é de fundamental importância para que as agências reguladoras exerçam suas funções e busquem atingir seus propósitos de forma eficiente. Para diminuir a assimetria de informações entre o governo e o operador, a agência de regulação precisa definir as informações de que necessita. Também é essencial definir procedimentos que resultem numa coleta de dados precisos, suficientes, confiáveis e com níveis de desagregação adequados e que perfaçam as atividades de análise, tratamento, manutenção e uso dessas informações. (World Bank, 2004)

No desenvolvimento de seus sistemas de informações, as agências reguladoras precisam considerar alguns aspectos:

a) oferecer aos cidadãos, governo e concessionários acesso as informações sobre as atividades da agência e das empresas prestadoras de serviços públicos;
b) promover a transparência nos processos regulatórios;
c) facilitar a interação pública com a agência;
d) proteger as informações sobre clientes e empresas que precisam ser mantidas privadas;
e) garantir que as informações relevantes sejam guardadas e recuperadas acuradamente e eficientemente; e

f) oferecer meios eficientes de baixo custo para as empresas suprirem a agência de informações. (World Bank, 2004)

A utilização de computador, de software e de recursos da internet e email permite o uso de tecnologia e desenvolvimento de ferramentas que proporcionam a agência maior facilidade, eficiência e segurança para atingir esses objetivos. A questão chave é como proteger as informações sobre usuários, governo e empresas que deveriam ser mantidas privadas e como divulgar informações de interesse público de forma eficaz e de baixo custo. (World Bank, 2004)

As informações – de empresas, usuários, mercado, etc. – propiciam a elaboração de séries temporais, estudos, análises e pesquisas, almejando medir e verificar constantemente as condições e o progresso do serviço prestado. Nestas situações, a qualidade da informação é de grande importância, o volume e o excesso de detalhe raramente ajudam e pode ser embaraçoso. A informação é vital para alcançar uma regulação efetiva, cabendo a agência questionar as concessionárias sobre uma ampla variedade de informações, cobrindo todos os aspectos dos negócios das empresas prestadoras de serviço. Essas informações permitem a agência monitorar e expor a performance do setor regulado. (OFWAT, 1995; WICS, 2006)

Desta forma, este estudo com base na metodologia de estudo de caso, buscando analisar um processo de desenvolvimento de software, aborda a seguir o tema, o problema, a justificativa, os objetivos e metodologia do estudo. Na revisão da literatura são apresentados os aspectos históricos do computador e do software, as metodologias de programação e mais detidamente o paradigma orientado a objeto. Na seqüência, os resultados do estudo de caso da agência de regulação, a AGERGS, da avaliação da linguagem Visual BASIC e do Sistema de Ouvidoria Pública. E, finalmente, as considerações finais.

2 TEMA

O progresso na área de software está estreitamente ligado a evolução do hardware. O lançamento no mercado de novas tecnologias, com melhor desempenho, menor tamanho e custos mais baixos, propiciam o desenvolvimento de sistemas baseados em computador mais sofisticados. Um produto de software não é mais considerado apenas como um programa de computador. Além deste, ele normalmente contém toda a documentação associada e as informações de instalação e configuração necessárias para que o programa opere corretamente.

O processo de desenvolvimento de software é o conjunto de atividades que geram um produto de software. A estruturação das atividades, os níveis de detalhamento, os prazos e os resultados são diferentes. Não existe um processo ideal, as organizações desenvolvem abordagens distintas para o desenvolvimento de software. Todavia, alguns processos são mais adequados do que outros, dependendo do tipo de aplicação. Se um processo inadequado for utilizado, isso provavelmente reduzirá a qualidade ou a utilidade do produto de software a ser desenvolvido.

Um processo de software pode envolver o desenvolvimento de um software desde o seu início, entretanto tem sido mais freqüente o caso de um software novo ser desenvolvido a partir da expansão e modificação de sistemas já existentes. Independentemente da área de aplicação, tamanho do projeto ou complexidade, um processo de software apresenta três fases genéricas: definição, desenvolvimento e manutenção.

Os passos mais recentes na evolução do desenvolvimento de software é desenvolvimento de projeto orientado a objeto. A metodologia orientada a objeto inicia-se com a abstração de dados, a qual encapsula o processamento com objetos de dados e oculta o acesso a eles, adicionando herança e vinculação dinâmica de tipos. Herança é um conceito poderoso que aumenta muito a reutilização potencial do software existente, oferecendo, portanto, a possibilidade de significativos aumentos na produtividade de desenvolvimento de software.

3 PROBLEMA

A Agência Estadual de Regulação dos Serviços Públicos Delegados do Rio Grande Do Sul – Agergs é uma agência de regulação criada, em 1997, na forma de autarquia e dotada de autonomia financeira, funcional e administrativa, com sede na Capital do Rio Grande do Sul. Seu objetivo é garantir a qualidade dos serviços públicos concedidos e o equilíbrio econômico e financeiro dos contratos.

A Ouvidoria da Agergs conta atualmente com o software SOA – Sistema de Ouvidoria da Agergs para organizar algumas de suas atividades como por exemplo, atender os usuários em suas reclamações, solicitações, ou demandas, bem como prestar informações e esclarecimento à população em geral. Desenvolvido internamente, o SOA atualmente precisa se adaptar e evoluir para atender as novas exigências dos processos administrativos do setor.

4 JUSTIFICATIVA

Não é improvável que o presente trabalho possa tanto incorporar a referência do Núcleo de Informática da Agergs, quanto instigar o aprimoramento do processo de desenvolvimento de software do setor. Consequentemente, pode resultar na melhoria da eficiência desses processos, elevar a produtividade e a qualidade do software, alterando positivamente a satisfação dos setores da agência, usuários finais dos produtos de software.

5 OBJETIVOS

5.1 OBJETIVO GERAL

Analisar o processo de desenvolvimento de software do Núcleo de Informática da Agergs.

5.2 OBJETIVOS ESPECÍFICOS

1. Investigar as principais metodologias de desenvolvimento de software;
2. Elaborar um estudo sobre programação orientada a objeto;
3. Identificar quais as metodologias de desenvolvimento de software a linguagem de programação Visual BASIC oferece suporte;
4. Identificar o paradigma de desenvolvimento empregado no SOA, Sistema de Ouvidoria da Agergs, desenvolvido pelo Núcleo de Informática da Agergs.

6 METODOLOGIA

O presente trabalho realiza um estudo de caso na organização da AGERS, envolvendo principalmente a Ouvidoria e o Núcleo de Informática (NIN).

O levantamento de dados é realizado com base nos recursos do NIN, tais como: acesso à rede de computadores, utilização de ferramentas de desenvolvimento de software disponíveis, de códigos fonte de programas, das domentações existentes e base de dados, entre outros que se façam necessários e diretamente relacionados ao presente estudo.

A obtenção dos resultados se faz por meio da coleta de informações com intuito de identificar o paradigma de desenvolvimento, a avaliar a linguagem Visual Basic e analisar o software SOA utilizando-se os recursos já descritos.

7 REVISÃO DA LITERATURA

Os primeiros computadores surgiram na década de 1940, foram construídos com base na tecnologia de válvulas eletrônicas. Com o desenvolvimento tecnológico, os computadores evoluíram utilizando novas tecnologias: as de válvulas foram substituídas pela tecnologia de transistores, que por sua vez pela de circuitos integrados. Além disso, eles sofreram um processo de miniaturização dos componentes e redução de custos. Em meados da década de 1970 surgem os microcomputadores fabricados em escala comercial. (Meirelles, 1994)

O desenvolvimento do software (programa) acompanhou a evolução do hardware (equipamentos e máquinas). Os primeiros computadores eram programados por meio de conexões utilizando fios. Com os conceitos de programa armazenado de von Neumann, os equipamentos passaram a ser programados em linguagem de máquina. Atualmente, os computadores estão sendo utilizados em diversas e infindáveis áreas de atividade. Por esse motivo, as linguagens de programação com propósitos e objetivos muito diferentes têm sido desenvolvidas. (Sebesta, 2000)

7.1 EVOLUÇÃO DOS COMPUTADORES

A evolução dos computadores é normalmente classificada em gerações. Existe um certo consenso na definição das três primeiras, mas historiadores e autores de informática não chegaram a um acordo quanto às datas de início da primeira geração e de término da terceira. Consequentemente, para alguns ainda estamos na terceira geração e para outros já entramos na quinta. No entanto, os acontecimentos que marcaram os avanços na tecnologia são mais relevantes do que os rótulos da classificação por gerações. (Meirelles, 1994)

Os primeiros computadores eletrônicos digitais surgiram na década de 1930. Os computadores da **primeira geração** são todos baseados em tecnologias de

válvulas eletrônicas. Essa geração vai até 1959, mas seu início é classificado entre 1942 e 1951. Os computadores da primeira geração normalmente quebravam após não muitas horas de uso. Tinham dispositivos de Entrada/Saída primitivos, calculavam com uma velocidade só de milésimo de segundo e eram programados em linguagem de máquina. Somente em 1951 surgiram os primeiros computadores produzidos em escala comercial. (Meirelles, 1994)

Nos equipamentos de **segunda geração,** a válvula foi substituída pelo transistor, tecnologia usada entre 1959 e 1965. Seu tamanho era cem vezes menor que o da válvula, não precisava de tempo para aquecimento, consumia menos energia, era mais rápido e mais confiável. Os computadores da segunda geração já calculavam em milionésimos de segundo (microssegundos). Em 1963, foi produzido o primeiro minicomputador com sucesso comercial. (Meirelles, 1994)

A **terceira geração** começa com a substituição dos transistores pela tecnologia de **Circuitos Integrados (CI)** e os computadores já calculavam em bilionésimos de segundo (nanossegundos). Os circuitos integrados são transistores e outros componentes eletrônicos miniaturizados e montados num único chip. Os modelos básicos dessa geração apresentavam várias opções e expansão e realizavam mais de 2 milhões de adições por segundo e cerca de 500 mil multiplicações. Esse fato tornou seus antecessores totalmente obsoletos e possibilitou a comercialização de bem mais de 30.000 sistemas. (Meirelles, 1994)

O CI entrou no mercado em 1959, mas só a partir de 1965 começou a substituir o transistor em computadores comercializados. As características dos CI resumem a evolução e tendência até os dias de hoje dos computadores: são muito mais confiáveis, mais rápidos, não têm partes móveis, os componentes são miniaturizados, os equipamentos são mais compactos, são de baixo consumo de energia e de menor custo. (Meirelles, 1994)

De 1970 em diante, as evoluções tecnológicas concentram-se principalmente na procura de processos mais precisos de miniaturização dos componentes internos dos computadores. Cada avanço alcançado em termos de tecnologia relaciona-se com a **escala de integração** (quantidade de circuitos que se pode colocar num único chip, CI). (Meirelles, 1994)

Entre 1965 e 1975 ocorre um desenvolvimento significativo dos mini computadores, respondendo à demanda por máquinas de menor porte e processamento menos centralizado. (Meirelles, 1994)

A terceira geração começa em 1965 e para alguns historiadores vai até os dias de hoje, uma vez que eles encaram o processo de 1965 em diante como uma evolução extremamente rápida mas natural. Outros autores localizam a **quarta geração** de 1970 ou 1971 em até hoje – considerando a importância de uma maior escala de integração alcançada pelos CI. Finalmente, a outra corrente usa o mesmo argumento da anterior, mas considerando que a miniaturização de fato ocorreu a partir de 1975 com o advento dos microprocessadores e dos microcomputadores. (Meirelles, 1994)

Em suma, a primeira geração era fabricada com válvulas, a segunda com transistores, da terceira em diante com CI de escalas de integração crescentes. (Meirelles, 1994)

Todos os computadores têm uma característica em comum, são máquinas de processamento serial com uma única Unidade Central de Processamento (CPU – *Central Processing Unit*), a qual faz o processamento propriamente dito. As operações são realizadas uma de cada vez copiando informação de um local selecionado da memória trazendo-a para a CPU que, de alguma forma, a modifica e tipicamente a armazena de volta na memória. (Meirelles, 1994)

O ano de 1977 é um marco importante na história da informática. Foi quando surgiram os microcomputadores fabricados em escala comercial, cujo microprocessador custava na época um milésimo do preço de uma CPU em 1960. De tamanho e preço muito reduzidos tinha uma capacidade semelhante às CPU de poucos anos atrás. (Meirelles, 1994)

Com essa tecnologia permitindo reunir num único chip todos os elementos que compunham a CPU, no fim da década de 1960, foram fabricados os primeiros minicomputadores bem mais baratos que os computadores da época e com capacidade suficiente para uma extensa gama de aplicações. Nos anos seguintes a capacidade continuou a crescer e o tamanho e os custos a cair. E, essa ainda é a tendência atual. Logo após o aparecimento do microprocessador surgiram os microcomputadores. Esse barateamento possibilitou a várias empresas a utilização

de recursos informática, que antes não tinham condições de manter um Centro de Processamento de Dados (CPD). (Meirelles, 1994)

Durante a década de 1980 a imagem dos micros cresceu muito. No início, a maioria era cética, os grandes fabricantes demoraram a entrar nesse mercado e os analistas de sistemas costumavam olhar com desprezo para esses “brinquedos eletrônicos”. (Meirelles, 1994)

Atualmente, os microcomputadores estão presentes nas casas, escritórios e substituindo desde computadores de médio porte até algumas aplicações de maior porte. (Meirelles, 1994)

7.2 EVOLUÇÃO DO SOFTWARE

O contexto em que o software foi desenvolvido está estreitamente ligado a evolução dos computadores. O melhor desempenho de hardware, menor tamanho e custo mais baixo precipitaram o aparecimento de sistemas baseados em computadores mais sofisticados. A evolução do software também é freqüentemente classificada em gerações. (Pressman, 1995)

A 1ª Geração corresponde aos primeiros anos do desenvolvimento de sistemas computadorizados. O hardware sofreu contínuas mudanças, enquanto o software usava uma orientação em lote (*batch*) para a maioria dos sistemas. Notáveis exceções foram os sistemas interativos. O hardware dedicava-se à execução de um único programa para uma aplicação específica. O hardware era de propósito geral, enquanto o software era projetado sob medida para cada aplicação e tinha uma distribuição relativamente limitada. O produto software (programas desenvolvidos para serem vendidos) estava em sua infância. A maior parte do software era desenvolvida e usada pela própria pessoa ou organização, a documentação muitas vezes não existia. (Pressman, 1995)

A 2ª Geração estendeu-se de meados da década de 1960 até o final da década de 1970. A multiprogramação e os sistemas multiusuários introduziram novos conceitos de interação entre homem e máquina. As técnicas interativas propiciaram o surgimento de novas aplicações e maiores níveis de sofisticação em

software. Sistemas em tempo real podiam coletar, analisar e transformar dados de múltiplas fontes em milissegundos. Essa geração também foi caracterizada pelo uso do produto de software e pelo advento da *software house*. Programas para *mainframes* e minicomputadores foram distribuídos e colocados a venda no mercado. Os avanços da armazenagem *on-line* levaram a primeira geração de sistemas de gerenciamento de bancos de dados. (Pressman, 1995)

À medida que o número de sistemas baseados em computador crescia, as bibliotecas de software começaram a se expandir. As empresas compravam produtos de software e desenvolviam projetos internamente, resultando em milhares de linhas de código ou instruções. Todos esses programas e suas instruções precisavam ser corrigidos quando eram detectadas falhas, alterados conforme as novas exigências dos usuários ou adaptados a um novo hardware. Essas atividades foram chamadas coletivamente de **manutenção de software**. O esforço despendido na manutenção de software começou a elevar os custos. A natureza personalizada de muitos programas tornava-os virtualmente impossíveis de sofrer manutenção. Como conseqüência houve o que passou a ser denominado de “crise de software”. (Pressman, 1995)

A 3ª Geração começou em meados da década de 1970 e continua até hoje. Os sistemas distribuídos (múltiplos computadores, cada um executando funções concorrentemente e comunicando-se um com o outro), as redes locais, as comunicações digitais de banda larga, a internet e a crescente demanda de acesso “instantâneo” a dados aumentaram consideravelmente a complexidade dos sistemas baseados em computador. (Pressman, 1995)

A quarta geração do software está apenas começando. As tecnologias orientadas a objetos estão rapidamente ocupando o lugar das abordagens mais convencionais para o desenvolvimento de software em muitas áreas de aplicação. (Pressman, 1995)

As linguagens de programação classificadas nas gerações são:

a) Primeira geração – Linguagem de Máquina;
b) Segunda geração – *Assembler*, Linguagem montadora;
c) Terceira geração – Linguagem de alto nível orientada para procedimentos, linguagens simbólicas;

d) Quarta geração – Linguagens orientadas para um problema ou aplicação. (Meirelles, 1994)

As linguagens da quarta geração ou de altíssimo nível podem ser classificadas em vários tipos como as de consulta (*query*), os geradores de aplicações, as chamadas linguagens descritivas e outras subdivisões mais recentes como: processadores de texto, planilhas eletrônicas, gerenciadores de banco de dados, gráficos, gerenciadores de comunicação, etc. No entanto, ainda não existe um consenso quanto à existência ou não da classificação dessa última geração. (Meirelles, 1994)

7.3 ARQUITETURA DO COMPUTADOR

A arquitetura básica dos computadores, chamada de arquitetura **von Neumann**, em homenagem a um de seus criadores, John von Neumann, é utilizada em quase todos os computadores digitais construídos a partir da década de 1940. Fato que teve enorme influência sobre o projeto das linguagens de programação, chamadas de **imperativas**. Com base nessa arquitetura é que a maioria das linguagens mais populares, desde a década de 1960, foi projetada e desenvolvida. (Sebesta, 2000)

Em um computador de von Neumann a CPU é separada da memória. Tanto os dados como os programas são armazenados na mesma memória. As instruções e os dados devem ser enviados da memória para a CPU, a qual de fato executa as instruções. Os resultados das operações obtidos na CPU devem ser novamente transferidos para a memória. Como as instruções, na maioria das vezes, podem ser executadas mais rapidamente do que transferidas ao processador para serem executadas, a velocidade da conexão entre a memória e o processador normalmente determina a velocidade do computador. Essa conexão é chamada de gargalo de von Neumann. Ele é o principal fator limitante da velocidade dessa arquitetura. (Sebesta, 2000)

Por causa da arquitetura von Neumann, os recursos centrais das linguagens imperativas são as variáveis, que modelam as células de memória e as instruções de atribuição, bem com têm por base a operação de canalização (*piping*) e a forma iterativa de repetição (o método mais eficiente dessa arquitetura). Os operandos das expressões são transmitidos da memória para a CPU, e o resultado da avaliação é transferido de volta para a célula da memória representada pelo lado esquerdo da atribuição. A iteração é rápida nos computadores de von Neumann porque as instruções são armazenadas em células adjacentes da memória. Essa eficiência desencoraja o uso da recursão para repetição, não obstante freqüentemente ser mais natural. (Sebesta, 2000)

7.4 DOMÍNIOS DA PROGRAMAÇÃO

Os primeiros computadores digitais, que surgiram na década de 1940, eram usados e, de fato, foram inventados para aplicações científicas. Tipicamente, as aplicações científicas têm estruturas de dados simples, mas exigem um grande número de computações aritméticas com ponto-flutuante. As estruturas de dados mais comuns são as matrizes e as estruturas de controle mais comuns são os laços de contagem e de seleção. As linguagens de programação de alto nível, inventadas para aplicações científicas, foram projetadas para suprir essas necessidades. A concorrente delas foi a linguagem *assembly* (linguagem de montagem) desse modo a eficiência era a primeira preocupação. A primeira linguagem para aplicações científicas foi o FORTRAN (*FORmula TRANslator*). O ALGOL 60 (*ALGOrithmic Language*) e a maioria das suas descendentes também se destinam a serem usadas nessa área, ainda que tenham sido projetadas também para outras áreas relacionadas. (Sebesta, 2000)

A linguagem de programação COBOL (*COmmon Business Oriented Language* – Linguagem Orientada aos Negócios) foi projetada para as aplicações comerciais com o objetivo principal em desenvolver sistemas comerciais, financeiros e administrativos para empresas e governos. (Sebesta, 2000)

Inteligência Artificial é uma área abrangente das aplicações de computador caracterizada pelo uso de computações simbólicas em vez de numéricas. A primeira linguagem de programação desenvolvida para aplicações de Inteligência Artificial amplamente utilizada foi a funcional LISP (*LISt Processor*), que surgiu em 1959. No início da década de 1970, surgiu uma abordagem alternativa para tais aplicações – a programação lógica. Uma linguagem de programação lógica é um exemplo baseado em regras. O Prolog (do francês: *PROgrammation en LOGique*) é a mais popular linguagem de programação lógica. (Sebesta, 2000)

Programação de sistemas operacionais e todas as ferramentas de suporte à programação de um computador são coletivamente conhecidos como seu software básico que é usado quase continuamente e, portanto, deve ter eficiência na execução. Os mais populares são os sistemas operacionais Linux e Windows. (Sebesta, 2000; Wikipédia, 2007)

Uma grande quantidade de linguagens de propósitos especiais foram desenvolvidas, como por exemplo: RPG (*Report Program Generator*) usada para produzir relatórios comerciais; APT (*Automatically Programmed Tool*) usada para instruir ferramentas de máquina programáveis; GPSS (*General Purpose Simulation System*) usada para simulação de sistemas; etc. (Sebesta, 2000; Wikipédia, 2007)

7.5 CRITÉRIOS DE AVALIAÇÃO DA LINGUAGEM

Os critérios considerados como mais importantes na avaliação dos recursos das linguagens de programação, quanto ao seu impacto no processo de desenvolvimento, inclusive manutenção, de software são: legibilidade, capacidade de escrita, confiabilidade e custo. Legibilidade é a facilidade com que os programas podem ser lidos e entendidos. Capacidade de escrita (*Writability*) é uma medida de quão facilmente uma linguagem pode ser usada para criar programas para um domínio de problema escolhido. A maioria das características da linguagem que afeta a legibilidade também afeta a capacidade de escrita. Isso se segue diretamente do fato de que escrever um programa exige uma releitura freqüente da

parte que já foi escrita pelo programador. Um programa é confiável se ele se comportar de acordo com suas especificações sob todas as condições. O custo final de uma linguagem de programação é uma função de muitas de suas características. De todos os fatores que contribuem para os custos da linguagem, três são os mais importantes: desenvolvimento do programa, manutenção e confiabilidade. Uma vez que esses são funções da legibilidade e da capacidade de escrita, os dois últimos critérios de avaliação são, por sua vez, os mais importantes. (Sebesta, 2000)

O estudo das linguagens de programação, à semelhança do estudo das naturais, pode ser dividido em exames da sintaxe e da semântica. A **sintaxe** de uma linguagem de programação é a forma de suas expressões, instruções e unidades de programa. Sua semântica é o significado destes três. A sintaxe ou a forma dos elementos de uma linguagem têm um efeito significativo sobre a legibilidade dos programas. (Sebesta, 2000)

A maioria dos critérios, especialmente a legibilidade, a capacidade de escrita e a confiabilidade não é nem precisamente definida, nem exatamente mensurável. Eles são conceitos úteis, fornecendo valiosas avaliações sobre o projeto e sobre as linguagens de programação. (Sebesta, 2000)

7.6 METODOLOGIAS DE PROGRAMAÇÃO

No início os computadores eram programados por meio de fios que interligavam as várias partes lógicas do equipamento. Mais tarde, com os conceitos de programa armazenado de von Neumann, os equipamentos passaram a permitir a programação em linguagem de máquina. Na década de 1950 foram introduzidas as linguagens simbólicas – o *Assembly*. A partir da década de 1960, os computadores passaram a ser programados com as chamadas linguagens **orientadas para o processo**. No final dessa década e o início da década de 1970 houve um intenso trabalho de análise tanto do processo de desenvolvimento de software quanto do projeto de linguagens de programação. Essa pesquisa foi provocada pela significativa mudança na relação de custo entre hardware e software, com redução

no custo dos equipamentos e elevação no custo dos programadores. O aumento de produtividade do programador foi relativamente pequeno. Enquanto, o software tornava-se progressivamente maior, mais complexo e de difícil manutenção. (Sebesta, 2000)

Verificação de tipos incompleta e instruções de controle insuficientes, que exigiam uso excessivo de *gotos*, foram as principais deficiências das linguagens de programação descobertas pelas pesquisas da década de 1970. Esses esforços resultaram em novas metodologias de desenvolvimento de software, adotando critérios de desenvolvimento de programas modulares, métodos *top-down* e refinamento passo a passo. No final dessa década, iniciou-se uma mudança das metodologias de projeto de **programas orientadas para o processo** para as de **programas orientadas a dados**. Expresso de maneira simples, os métodos orientados a dados enfatizam o projeto de dados, fazendo uso de tipos de dados abstratos na resolução de problemas. (Meirelles, 1994; Sebesta, 2000)

As linguagens de programação necessariamente devem oferecer suporte para abstração de dados, para que esta seja eficientemente implementada no projeto de sistemas de software. A primeira a oferecer um limitado suporte para abstração de dados foi a SIMULA 67, a qual foi apresentada em 1967. Alguns conceitos que essa linguagem introduziu tornaram-se importantes, mesmo que ela jamais tenha obtido um uso generalizado. Os benefícios da abstração de dados não foram amplamente reconhecidos até o início da década de 1970. No entanto, desde o final dessa década, a maioria das linguagens de programação desenvolvidas suporta abstração de dados. (Sebesta, 2000)

Como uma evolução do desenvolvimento de software orientado a dados, surge em meados da década de 80 a **programação orientada a objeto**. A metodologia orientada a objeto faz uso não apenas da abstração de dados, a qual encapsula o processamento com objetos de dados, ocultando o acesso a eles, como também dos conceitos de herança e de vinculação dinâmica de tipos. A herança é um conceito poderoso que permite a reutilização de software, o que viabiliza um aumento significativo na produtividade de desenvolvimento de software. Fato relevante para o aumento de popularidade das linguagens orientadas a objeto.

A vinculação dinâmica de tipos permite um uso mais flexível da herança. (Sebesta, 2000)

A programação orientada a objeto desenvolveu-se juntamente com a linguagem que deu suporte para os seus conceitos: a Smalltalk, apresentada em 1972. Embora essa linguagem não tenha sido muito utilizada como algumas outras, o suporte para programação orientada a objeto atualmente faz parte das linguagens de programação mais populares, inclusive da Ada 95, do Java e do C++. A programação orientada ao processo é, em certo sentido, o oposto da programação orientada a dados. Embora os métodos orientados ao processo não tenham sido abandonados, os métodos orientados a dados agora dominam o desenvolvimento de software. (Sebesta, 2000)

7.7 LINGUAGENS DE PROGRAMAÇÃO

7.7.1 Categorias de Linguagem

As linguagens de programação muitas vezes são classificadas em quatro categorias: imperativas, funcionais, lógicas e orientadas a objeto. As linguagens de programação orientadas a objeto mais populares desenvolveram-se a partir das imperativas. As características das linguagens imperativas já foram discutidas, também já foi descrito como as linguagens de programação orientadas a objeto mais populares desenvolveram-se a partir das imperativas. Não obstante o paradigma de desenvolvimento de software diferir bastante do paradigma orientado para procedimentos normalmente usado com as linguagens imperativas, as extensões a uma linguagem imperativa necessárias para suportar a programação orientada a objeto não são opressivas. Por exemplo, as expressões, as instruções de atribuição e as instruções de controle do C e do Java são quase idênticas; por outro lado, as matrizes, os subprogramas e a semântica da linguagem Java são muito diferentes em C. (Sebesta, 2000)

Uma linguagem de programação lógica é um exemplo baseado em regras. Em uma linguagem imperativa, um algoritmo é especificado com grandes detalhe, e a ordem de execução específica das instruções ou dos comandos deve ser incluída. Em uma linguagem baseada em regras, estas são especificadas sem nenhuma ordem particular, e o sistema de implementação deve escolher uma ordem de execução que produza o resultado desejado. Essa abordagem ao desenvolvimento de software é radicalmente diferente daquelas usadas com os outros três tipos de linguagens, e, evidentemente, exige um tipo de linguagem completamente diferente. O Prolog é a mais popular linguagem de programação lógica. (Sebesta, 2000)

7.7.1.1 Linguagem funcional

Uma linguagem funcional ou aplicativa é aquela cujo principal meio de fazer computações é aplicando funções a determinados parâmetros. A programação pode ser feita em uma linguagem funcional sem o tipo de variáveis usadas nas imperativas, sem instruções de atribuição e sem iteração. Ainda que muitos cientistas da computação tenham feito exposições sobre os inúmeros benefícios das linguagens funcionais, como o LISP, é improvável que eles deixem de lado as imperativas até que um computador não-von Neumann seja projetado, permitindo a execução eficiente de programas em linguagens funcionais. (Sebesta, 2000)

As máquinas de arquitetura paralela que surgiram nas últimas décadas, apresentam alguma promessa de agilizar a velocidade de execução de programas funcionais, mas até agora isso não tem sido suficiente para torná-las competitivas com programas imperativos. (Sebesta, 2000)

O elevado grau de similaridade entre as linguagens imperativas provém, em parte, de uma de suas bases comuns de projeto: a arquitetura de von Neumann,. Podemos pensar nas linguagens imperativas coletivamente como uma progressão de desenvolvimentos para melhorar o modelo básico, o FORTRAN 1. Tudo foi projetado para um uso eficiente de computadores com a arquitetura de von

Neumann. Não obstante o estilo imperativo de programação ter sido considerado aceitável pela maioria dos programadores, o gasto de recorrer fortemente à arquitetura subjacente é visto, por alguns, como uma restrição desnecessária ao processo de desenvolvimento de software. (Sebesta, 2000)

Existem outras bases de projeto de linguagem, muitas das quais orientadas mais a paradigmas e as metodologias de programação particulares do que à execução eficiente em uma arquitetura de computador particular. Até agora, entretanto, a reduzida eficiência ao executar programas escritos nessas linguagens tem impedido que elas se tornem tão populares quanto às linguagens imperativas. (Sebesta, 2000)

O paradigma de programação funcional fundamentado em funções matemáticas é a base de projeto dos estilos de linguagens imperativas mais importantes. Esse estilo de programação é suportado pelas linguagens de programação funcionais ou aplicativas. (Sebesta, 2000)

Um grande número de linguagens de programação funcionais foi desenvolvido. O LISP é a mais antiga e iniciou-se como uma linguagem puramente funcional, mas logo teve um grande número de recursos de linguagem imperativa adicionados a fim de aumentar sua eficiência e sua facilidade de uso. Ele ainda é a mais importante das linguagens funcionais. Pelo menos em termo de ser a única que conseguiu obter um uso generalizado. (Sebesta, 2000)

O objetivo do projeto de uma linguagem de programação funcional é imitar as funções matemáticas no maior grau possível. Isso resulta em uma abordagem à solução de problemas que difere fundamentalmente dos métodos usados com as linguagens imperativas. (Sebesta, 2000)

Uma linguagem de programação puramente funcional não usa variáveis ou instruções de atribuição. Isso libera o programador de preocupar-se com as células de memória do computador no qual o programa é executado. Sem variáveis, construções iterativas não são possíveis, porque elas são controladas por variáveis. A repetição deve ser feita por meio de recursão, não por meio de laços. (Sebesta, 2000)

As linguagens de programação funcionais seguem o modelo das funções matemáticas. Em sua forma pura elas não usam variáveis ou instruções de

atribuição para produzir resultados; do contrário, usam aplicações funcionais, expressões condicionais e resurção para controle de execução, e forma funcionais para construir funções complexas. (Sebesta, 2000)

Ainda que possa haver vantagens nas linguagens puramente funcionais em relação à suas parentes imperativas, sua menor eficiência de execução nas máquinas de von Neumann tem impedido que elas sejam consideradas por muitos como substitutas. (Sebesta, 2000)

7.7.1.2 Linguagem Lógica

No paradigma da programação lógica, a abordagem é expressar programas na forma de lógica simbólica e usar um processo de inferência lógica para produzir resultados. Os programas lógicos são declarativos em vez de baseados em procedimentos, o que significa que somente as especificações dos resultados desejados são declaradas em vez de procedimentos detalhados para produzi-los. (Sebesta, 2000)

A programação que usa uma forma de lógica simbólica como linguagem freqüentemente é chamada **programação lógica** e as linguagens baseadas na lógica simbólica são chamadas **linguagens de programação lógicas** ou **linguagens declarativas**. (Sebesta, 2000)

A sintaxe das linguagens de programação lógica são notavelmente diferentes da sintaxe das linguagens imperativas e das funcionais. A semântica dos programas lógicos também sustenta pouca semelhança com a dos programas em linguagem imperativa. (Sebesta, 2000)

A lógica simbólica constitui a base para a programação lógica e para as linguagens de programação lógica. A abordagem de programação lógica é usar, como banco de dados, uma coleção de fatos e de regras que declaram relações entre fatos, e usar um processo de inferência automático para verificar a validade de novas proposições, supondo que os fatos e as regras do banco de dados sejam

verdadeiros. Essa abordagem é desenvolvida para demonstração automática de teoremas. (Sebesta, 2000)

As origens da programação lógica situam-se no desenvolvimento da regra de resolução para inferência lógica, em meados da década de 1960. O Prolog é a linguagem de programação lógica mais popular e de uso generalizado. As instruções Prolog são fatos, regras ou metas. A maioria é composta de estruturas de proposições atômicas e de operadores lógicos, não obstante expressões aritméticas também sejam permitidas. (Sebesta, 2000)

Há uma série de problemas com a situação atual da programação lógica. Por razões de eficiência, e até mesmo para evitar laços infinitos, às vezes os programadores precisam declarar informações de fluxo de controle em seus programas. Além disso, há os problemas de pressuposição de um mundo fechado e de negação. (Sebesta, 2000)

A programação lógica tem sido usada em muitas áreas diferentes principalmente em sistemas de bancos de dados relacionais, em sistemas especialistas e no processamento de linguagem natural. (Sebesta, 2000)

7.7.1.3 Linguagem de Marcação

As linguagens de marcação (*markup*) nada mais são do que um padrão para a definição de vínculos de hipertexto entre documentos. O HTML (*HyperText Markup Language*) é uma linguagem de marcação de uso generalizado na elaboração de páginas para a internet. Muitas vezes essas linguagens são confundidas com as linguagens de programação. No entanto, as linguagens de marcação não especificam computações, muito pelo contrário, elas descrevem a aparência geral de documentos. (Sebesta, 2000)

7.7.2 Ambiente de Programação

Um ambiente de programação é um conjunto de ferramentas usadas no desenvolvimento de software. (Sebesta, 2000)

O UNIX é o mais antigo ambiente de programação, distribuído pela primeira vez na década de 1970. Ele foi construído em torno de um sistema operacional portável com compartilhamento de tempo (*time-sharing*), além de fornecer um amplo conjunto de ferramentas de apoio a produção e manutenção de software em uma variedade de linguagens. (Sebesta, 2000)

O Borland C++ é um ambiente de programação que roda em microcomputadores. Ele oferece um compilador integrado, um editor, um depurador e um sistema de arquivos, em que todos os quatro são acessados por meio de uma interface gráfica. Um recurso conveniente desse tipo de ambiente é que, quando o compilador encontra um erro de sintaxe, ele faz uma parada e muda para o editor, deixando o cursor no ponto do programa-fonte em que o erro foi detectado. (Sebesta, 2000)

O Smalltalk é uma linguagem e um ambiente de programação integrados, mas ela é mais elaborada, complexa e poderosa do que o Borland C++. O Smalltalk foi o primeiro a fazer uso de um sistema de janelas e um dispositivo de indicação por mouse para oferecer ao usuário uma interface uniforme a todas as ferramentas. (Sebesta, 2000)

O passo mais recente na evolução dos ambientes de desenvolvimento de software é representado pelo Microsoft Visual C++. É uma grande e elaborada coleção de ferramentas de desenvolvimento de software, todas usadas por uma interface provida de janelas. Esse sistema, juntamente com outros sistemas similares como o Visual BASIC, o Delphi e o Java Development Kit, da Sun Microsystems, oferecem maneiras simples de construir interfaces gráficas para os programadores. (Sebesta, 2000)

7.8 SISTEMAS BASEADOS EM COMPUTADOR

Nos sistemas baseados em computador, o software tomou lugar do hardware como o elemento de sistema mais difícil de planejar e administrar, com menos probabilidade de obter sucesso em termos de cumprimento de prazos e custos. Contudo, a demanda por software não diminui e esses sistemas crescem em número, complexidade e aplicação. (Pressman, 1995)

Um **sistema baseado em computador** é um conjunto ou disposição de elementos que é organizado para executar certo método, procedimento ou controle ao processar informações. Esses elementos se combinam de muitas maneiras para transformar informações. Os elementos desse sistema freqüentemente incluem software, hardware, pessoas, banco de dados, documentação e procedimentos. O software são os programas de computador, estruturas de dados e documentação correlata que servem para efetivar o método, processo ou controle lógico necessário. O hardware são os dispositivos eletrônicos (CPU, memória, etc.) que fornecem capacidade ao computador, e os eletromecânicos (sensores, motores, etc.) que oferecem funções ao mundo externo. As pessoas são os usuários e operadores de hardware e software. Os bancos de dados são uma grande e organizada coleção de informações a que se tem acesso pelo software e faz parte integrante da função do sistema. A documentação são os manuais, formulários e outras informações descritivas que retratam o uso e operação do sistema. Os procedimentos são os passos que definem o uso específico de cada elemento do sistema ou o contexto processual em que o sistema reside. (Pressman, 1995)

Em meados dos anos 1970, numa tentativa de contornar a crise do software e dar um tratamento de engenharia mais sistemático e controlado ao desenvolvimento de sistemas de software complexos, surgiu a Engenharia de Software. Um sistema de software se caracteriza por um conjunto de componentes abstratos de software (estruturas de dados e algoritmos) encapsulados na forma de procedimentos, funções, módulos, objetos ou agentes interconectados entre si, compondo a arquitetura do software, que deverão ser executados em sistemas computacionais. A Engenharia de Software é uma área do conhecimento que tem por princípio aperfeiçoar a organização, a qualidade e a produtividade dos

processos de especificação, desenvolvimento e manutenção de sistemas de software. (Wikepedia, 2007)

Os **métodos** de engenharia de software proporcionam os detalhes de “como fazer” para construir o software. Os métodos envolvem um amplo conjunto de tarefas que incluem: planejamento (estimativas de projeto), análise de requisitos de software e de sistemas, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção. (Pressman, 1995)

O ciclo de vida clássico do software abrange as seguintes etapas: análise, projeto, codificação, teste e manutenção. O processo de desenvolvimento de software contém três fases genéricas, independentemente do paradigma de engenharia de software escolhido. As três fases definição, desenvolvimento e manutenção são encontradas em todo desenvolvimento de software, independentemente da área de aplicação, tamanho do projeto ou complexidade. (Pressman, 1995)

7.9 DOMÍNIOS DA INFORMAÇÃO

O software é construído para processar **dados**: para transformar dados de uma forma em outra, ou seja, aceitar entrada, manipulá-los de alguma maneira e produzir saída. É importante observar que o software também processa eventos. Um **evento** representa algum aspecto do controle do sistema e, de fato, nada mais é do que os tipos de dados *boolean* (verdadeiro ou falso, ligado ou desligado, existente ou não). Portanto, **dados** (números, caracteres, imagens, sons etc.) e **controles** (eventos) residem no domínio da informação de um problema. (Pressman, 1995)

O **domínio da informação** encerra três diferentes pontos de vista sobre os dados e sobre o controle, quando cada um é processado por um programa de computador: fluxo de informação, conteúdo da informação, e estrutura da informação. O **fluxo de informação** representa a maneira pela qual os dados e o controle se modificam à medida que cada um se movimenta pelo sistema. A entrada

é transformada em informações intermediárias que são depois transformadas em saída. Ao longo desse caminho (ou caminhos) de informação, outras informações podem ser introduzidas a partir de um **depósito de dados** (*store*), por exemplo, um arquivo em disco ou *buffer* de memória. As transformações aplicadas aos dados são funções ou subfunções que um programa deve executar. Os dados e o controle que se movem entre duas transformações ou funções definem a interface de cada função. (Pressman, 1995)

O **conteúdo da informação** representa os itens de dados e os itens de controle individuais que compreendem certo item de informação mais amplo. Por exemplo, o item de dados de um cadastro é uma combinação de uma série de informações: nome, endereço, telefone, etc. Desse modo, o conteúdo do item cadastro é definido pelos itens que são necessários para criá-lo. Similarmente, o conteúdo de um item de controle denominado status do sistema poderia ser definido por uma cadeia de caracteres. Cada caracter representa um item de informação distinto que indica se um dispositivo particular está ou não conectado. (Pressman, 1995)

A **estrutura da informação** representa a organização interna de vários itens de controle e de dados. Deve-se observar que a estrutura de dados refere-se ao projeto e à implementação de estruturas de informação por meio do software. (Pressman, 1995)

7.10 METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Uma metodologia de engenharia de software é um processo para a produção organizada de software, com utilização de uma coleção de técnicas predefinidas e convenções notacionais. Uma metodologia costuma ser apresentada como uma série de etapas, com técnicas e notação associadas a cada etapa. As etapas da produção de software são habitualmente organizadas em ciclo de vida composto por diversas fases de desenvolvimento. O ciclo de vida completo de um software passa pela formulação inicial do problema, pela análise, projeto, implementação e pelos

testes do software, e é seguido por uma fase operacional durante a qual são executados a manutenção e o aperfeiçoamento. (Rumbaugh pg. 191)

O **desenvolvimento baseado em objetos** corresponde a um relativamente novo modo de tratar o software com base em abstrações que existem no mundo real. Neste contexto, **desenvolvimento** refere-se à parte inicial do ciclo de vida do software: análise, projeto, implementação. A essência do desenvolvimento baseado em objetos é a identificação e a organização de conceitos do domínio da aplicação. As linguagens de programação baseadas em objetos – mecanismos de implementação – são úteis para remover as restrições devidas à inflexibilidade das linguagens de programação tradicionais. (Rumbaugh et al.,1994)

7.10.1 Análise de Requisitos

A análise requisitos de software e de sistemas é o primeiro passo do processo de engenharia de software. É nesse ponto que o escopo do software é definido e suas especificações são descritas num documento que se torna a base para todas as fases seguintes. A análise preocupa-se com as funções, comportamentos e informações de um problema, criando modelos, dividindo o problema em partições e desenvolvendo representações que descrevem a essência dos requisitos e os detalhes de implementação. (Pressman, 1995)

Em muitos casos, não é possível especificar completamente um problema numa primeira etapa. A prototipação oferece uma abordagem alternativa que, com o uso de ferramentas e de técnicas específicas para este fim, resulta num modelo executável do software, a partir do qual as exigências podem ser refinadas. (Pressman, 1995)

O **modelo de software** ajuda a entender a informação, a função e o comportamento de um sistema, tornando a tarefa de análise de requisitos mais fácil e mais sistemática. O modelo concentra-se naquilo que o sistema deve fazer, não em como ele o faz. Ele deve corresponder a uma representação da informação que

o software transforma, das funções que viabilizam essa transformação e do comportamento do sistema quando essa transformação está sendo executada. Na construção de modelos pode ser feito uso de uma notação gráfica descrevendo as informações, o processamento, o comportamento do sistema e outras características. (Pressman, 1995)

Os domínios funcionais, comportamentais e de informação do software podem ser divididos em partições. O **particionamento** decompõe o problema em suas partes constituintes. Inicialmente, é feita uma representação hierárquica da função ou da informação. A seguir, divide-se em partições o elemento superior, ou expondo detalhes crescentes com um deslocamento verticalmente na hierarquia ou decompondo funcionalmente o problema com um deslocamento horizontalmente na hierarquia. (Pressman, 1995)

A **especificação de requisitos de software** é o resultado dessa fase de análise, sendo importante à execução de uma revisão para garantir que o desenvolvedor e o cliente tenham a mesma percepção do sistema. (Pressman, 1995)

7.10.2 Projeto e Implementação de Software

O projeto é o núcleo técnico da engenharia de software, que resulta em representações de software que podem ser avaliadas quanto à qualidade. Durante esta etapa, progressivos refinamentos da estrutura de dados, da arquitetura de programa e detalhes procedimentais são desenvolvidos, revisados e documentados. (Pressman, 1995)

O projeto de software pode ser visto de uma perspectiva técnica ou de uma perspectiva de administração de projetos. De uma perspectiva técnica, o projeto é composto de quatro atividades: projeto de dados, projeto arquitetural, projeto procedimental, e projeto de interfaces. Do ponto de vista administrativo, o projeto

desenvolve-se a partir do projeto preliminar e do projeto detalhado. (Pressman, 1995)

A notação de projeto, combinada com conceitos de programação estruturada, possibilita representar os detalhes procedimentais de forma que facilite a conversão em código. Notações gráficas, tabulares e textuais podem ser utilizadas. (Pressman, 1995)

A evolução do projeto de software é um processo contínuo que se estende há três décadas. Os primeiros trabalhos concentravam-se em critérios para o desenvolvimento de programas modulares e métodos *top-down* para se aprimorar a arquitetura de software. Os aspectos procedimentais de definição de projeto desenvolveram-se na direção de um paradigma denominado **programação estruturada**. O trabalho posterior propunha métodos para a tradução de fluxo de dados ou estrutura de dados numa definição de projeto. Abordagens mais recentes propõem uma abordagem **orientada a objeto**, à derivação de projetos. (Pressman, 1995)

Cada um desses métodos tem uma série de características em comum. Um mecanismo para a tradução da representação do domínio de informação numa representação de projeto. Uma notação para representar os componentes funcionais e suas interfaces. Heurísticas para refinamento e divisão em partições. E, diretrizes para a avaliação da qualidade. (Pressman, 1995)

Uma série de conceitos fundamentais de projeto de software foi desenvolvida ao longo dessas três décadas. A modularidade, tanto de programa como de dados, e o conceito de abstração possibilitam a simplificação e reuso dos componentes de software. O refinamento é um mecanismo que proporciona representações de sucessivas camadas de detalhes funcionais. As estruturas de programa e de dados contribuem para uma visão global da arquitetura de software, enquanto o procedimento oferece os detalhes necessários para a implementação algorítmica. A ocultação de informações e a independência funcional oferecem a heurística para se conseguir efetiva modularidade. (Pressman, 1995)

7.10.2.1 Aspectos Fundamentais do Projeto de Software

O projeto é o primeiro passo da fase de desenvolvimento de qualquer produto ou sistema de engenharia. Encontra-se no núcleo técnico do processo de engenharia de software e é aplicado independentemente do paradigma de desenvolvimento usado. (Pressman, 1995)

Iniciando-se tão logo os requisitos de software tenham sido analisados e especificados, o projeto é a primeira dentre as três atividades técnicas – projeto, codificação e teste – que são exigidas para se construir e verificar um software. Cada atividade transforma informações de um modo que resultará, por fim, num software de computador validado. (Pressman, 1995)

Os requisitos de software, indicados pelos modelos comportamentais, funcionais e de informação, abastecem a fase de projeto. Usando um determinado método, essa fase produz um projeto de dados, um projeto arquitetural e um projeto procedimental. **O projeto de dados** transforma o modelo do domínio de informação criado durante a análise nas estruturas de dados que serão exigidas para se implementar o software. O **projeto arquitetural** define o relacionamento entre os grandes componentes estruturais do programa. O **projeto procedimental** transforma os componentes estruturais numa descrição procedimental do software. O código-fonte é gerado e a atividade de testes é levada a efeito para integrar e validar o software. (Pressman, 1995)

As atividades de projeto, codificação e teste absorvem 75% ou mais dos custos da engenharia de software, excluindo-se a manutenção. A importância do projeto de software tem relação direta com a qualidade. O projeto é o processo onde a qualidade é fomentada durante o seu desenvolvimento. (Pressman, 1995)

7.10.2.2 Processo de Projeto

O projeto de software é o processo pelo qual os requisitos são traduzidos numa representação do software. Inicialmente, a representação descreve uma visão holística do software. Subseqüentes refinamentos levam a uma representação que está muito próxima ao código-fonte. (Pressman, 1995)

Do ponto de vista da administração, o projeto de software é levado a efeito em dois passos. O **projeto preliminar** preocupa-se com a transformação dos requisitos numa arquitetura de dados e de software. O **projeto detalhado** concentra-se nos aprimoramentos de representação estrutural que levam à representações algorítmicas e de estruturas de dados detalhadas. (Pressman, 1995)

Dentro do contexto do projeto preliminar e detalhado, uma série de diferentes atividades se desenvolve. Além dos projetos procedimental, arquitetural e de dados, muitas aplicações modernas têm uma atividade de projeto de interface distinta. O **projeto de interface** estabelece o formato (*layout*) e os mecanismos para a interação entre o usuário e o computador. (Pressman, 1995)

Desta forma, o projeto baseia-se em informações e os métodos de projeto de software são derivados da consideração do domínio de informação: orientado ao fluxo de dados, orientado a objetos e orientado a dados. (Pressman, 1995)

7.10.2.2.1 Projeto Orientado ao Fluxo de Dados

O projeto orientado ao fluxo de dados é um método que usa características de fluxo de informações para derivar uma estrutura de programa. Um diagrama de fluxo de dados leva à estrutura de programa usando-se uma dentre duas abordagens de mapeamento – análise das transformações e análise das transações. O resultado desta etapa corresponde a um projeto preliminar do software, com a definição dos módulos, das interfaces e da estrutura de dados.

Essas representações de projeto formam a base para todo o trabalho de desenvolvimento subseqüente. (Pressman, 1995)

7.10.2.2.2 Projeto Orientado a Objeto

O projeto orientado a objeto cria um modelo do mundo real que pode ser realizado em software. Os objetos oferecem um mecanismo para representar o domínio da informação, enquanto as operações descrevem o processamento que é associado ao domínio de informação. (Pressman, 1995)

7.10.2.2.3 Projeto Orientado a Dados

O projeto orientado para a estrutura de dados, como todos os grandes métodos de projeto de software, focaliza o domínio da informação. Porém, em vez de se concentrar no fluxo de dados, os métodos orientados para a estrutura de dados usam a estrutura de informações como a base para a derivação do projeto. (Pressman, 1995)

7.11 PROGRAMAÇÃO ORIENTADA A OBJETO

Orientação a Objetos é uma abordagem de programação que procura explorar a intuição. Os “átomos” da computação orientada, os **objetos**, são análogos aos objetos existentes no mundo físico. Isto produz um modelo de programação que difere de forma marcante da tradicional visão “funcional”. Essa diferença é, ao mesmo tempo, um ponto forte e uma fraqueza da bordagem

orientada a objetos. È um ponto forte devido ao apelo que gera à nossa intuição e também porque orientação a objetos se mostra produtiva tanto na teoria quanto na prática. Por outro lado, representa uma fraqueza, porque os métodos de desenvolvimento de software mais tradicionais não combinam com esta nova abordagem. Atualmente existem vários métodos diferentes, mas que são específicos para o desenvolvimento de software orientado a objetos. (Coleman et al., 1996)

No enfoque orientado a objetos, os átomos do processo de computação são os **objetos**, que trocam **mensagens** entre si. Estas mensagens resultam na ativação de **métodos**, os quais realizam a ações necessárias. O emissor da mensagem não precisa saber como o objeto receptor organiza o seu estado interno, mas apenas que este objeto responde a certas mensagens de maneira bem definida. (Coleman et al., 1996)

Os objetos, quando compartilharem uma única interface, são agrupados em **classes**, ou seja, respondem as mesmas mensagens da mesma maneira. Isso permite que vários objetos sejam descritos por apenas algumas classes. As classes são os blocos de construção da maior parte das linguagens orientadas a objetos. (Coleman et al., 1996)

Durante a execução do sistema, os objetos podem ser construídos, executar ações, serem destruídos ou se tornam inacessíveis. O modelo computacional é essencialmente dinâmico. (Coleman et al., 1996)

No enfoque mais tradicional orientado a funções, os átomos da computação são as **funções**, as quais atuam sobre um único **estado** compartilhado, possivelmente apresentando um alto grau de estruturação. Normalmente, o estado é estático, ou seja, a sua estrutura não se altera com o passar do tempo, apesar de, certamente, termos alterações nos valores armazenados na estrutura. Qualquer função pode atuar sobre qualquer parte do estado. Mesmo que o sistema possa ser dividido em módulos independentes, cada um deles tendo o controle sobre uma parte do estado, o enfoque orientado a funções não possui um modelo subjacente que diga como essa divisão seria realizada. . (Coleman et al., 1996)

Orientação a objetos já provou ser popular e eficaz. Foi incorporada em várias linguagens de programação, sendo as mais conhecidas: C++, Eiffel, Smalltalk e CLOS (*Common Lisp Object System*). (Coleman et al., 1996)

Segundo Coleman et al. (1996), as principais vantagens do desenvolvimento orientado a objetos são as seguintes:

Abstração de dados: Os detalhes referentes às representações das classes serão visíveis apenas a seus métodos; implementações diferentes de uma classe poderão ser utilizadas e sem alterações no código que utiliza a classe em questão. A abstração de dados não é uma característica exclusiva da tecnologia de objetos, mas surge naturalmente em seu contexto;

Compatibilidade: As heurísticas para a construção das classes e suas interfaces levam a componentes de software que são mais fáceis de serem combinados;

Flexibilidade: as classes, ou coleções de classes fortemente relacionadas, delimitam unidades naturais para a alocação de tarefas no desenvolvimento de software;

Extensibilidade: O software construído com o uso de técnicas orientadas a objetos tende a ser mais facilmente entendido. Há duas razões para isso: a herança permite que novas classes sejam construídas a partir de outras já existentes, sem deixar de participar de todos os relacionamentos originais; além disso, as classes formam uma estrutura fracamente acoplada, o que facilita as alterações;

Manutenção: A modularidade natural da estrutura de classe facilita o controle dos efeitos gerados pelas alterações, e o uso de herança diminui a quantidade conceitos díspares necessários ao entendimento do código;

Reutilização: O encapsulamento de métodos e representações de dados para a construção de classes facilita o desenvolvimento de software reutilizável.

Um dos benefícios proporcionados pela tecnologia orientada a objetos se evidencia no suporte técnico superior para a reutilização de componentes de software. A abordagem orientada a objetos disponibiliza vários mecanismos e técnicas que reduzem a complexidade inerente ao desenvolvimento de muitos

projetos de software. Alguns deles são: a modelagem do mundo real, o encapsulamento, a herança, o polimorfismo. . (Coleman et al., 1996)

Os conceitos básicos, segundo a Enciclopédia Livre – Wikipedia (2007), da orientação a objeto são:

Classe: representa um conjunto de objetos com características afins. Uma classe define o comportamento do objeto (por meio de procedimentos), e os estados que ele é capaz de manter (por meio de atributos), por exemplo: Pessoas;

Objeto: é uma instância de uma classe. Um objeto é capaz de armazenar os estados por meio de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. Exemplo de um objeto da classe Pessoas: Funcionário, Clientes, João, José, Maria;

Atributos: são os dados ou as informações do objeto, basicamente a estrutura de dados que vai representar a classe. Exemplo de atributos de Funcionário: nome, endereço, telefone, CPF, etc. ;

Métodos: definem as habilidades dos objetos. João é uma instância da classe Pessoas, portanto tem habilidade para andar, implementada através do método deUmPasso(). Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é invocado através do objeto, no caso João. Dentro do programa, a utilização de um método deve afetar apenas um objeto em particular; Todas as pessoas podem caminhar, mas você quer que apenas João dê um passo. Normalmente, uma classe possui diversos métodos, que no caso da classe Pessoas poderiam ser **sentar ()**, **comer ()** e **andar ();**

Mensagem é uma chamada a um objeto para invocar um de seus métodos, ativando um comportamento descrito por sua classe. Também pode ser direcionada diretamente a uma classe;

Sobrecarga: é a utilização do mesmo nome para símbolos ou métodos com operações ou funcionalidades distintas. Geralmente diferencia-se os métodos pela sua assinatura;

Herança: (ou generalização) é o mecanismo pelo qual uma classe (sub-classe) pode estender outra classe (super-classe), aproveitando seus comportamentos (métodos) e estados possíveis (atributos). Há Herança múltipla quando uma sub-classe possui mais de uma super-classe. Essa relação é

normalmente chamada de relação "é um". Um exemplo de herança: Mamífero é super-classe de Humano. Ou seja, um Humano é um mamífero;

Associação: é o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode tratar-se de uma associação simples "usa um" ou de um acoplamento "parte de". Por exemplo: Um humano usa um telefone. A tecla "1" é parte de um telefone;

Encapsulamento: consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados. Exemplo: você não precisa conhecer os detalhes dos circuitos de um telefone para utilizá-lo. A carcaça do telefone encapsula esses detalhes, provendo a você uma interface mais amigável – o aparelho de telefone;

Abstração: é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software;

Polimorfismo: é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma assinatura (lista de parâmetros e retorno). No entanto, esses métodos apresentam comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse;

Interface: é um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface.

7.12 DESENVOLVIMENTO ORIENTADO A OBJETO

Vários métodos de análise e projeto orientados a objetos estão documentados na literatura ou são oferecidos por consultores e companhias CASE (*Computer-Aided Software Engineering*). Em 1992, o *Object Management Group*

iniciou um estudo comparativo englobando mais de vinte métodos. A maior parte desses métodos envolve variações de idéias, tais como, modelagem de objetos com entidade-relacionamento, máquinas de estados, cenários, etc. Na literatura encontram-se descritas várias comparações entre as metodologias orientadas a objetos, mas não há uma maneira completamente aceita para a comparação ou avaliação de métodos diferentes. (Coleman et al., 1996)

Os métodos mais importantes são: Coad-Yourdon, OMT (*Object Modeling Technique*), Booch, Objectory, Métodos Formais, CRC e Fusion. Outros métodos importantes são: Martin-Odell, OSA e Shkaer-Mellor. (Coleman et al., 1996)

O método Coad-Yourdon é destinado primariamente ao desenvolvimento de sistemas de gerenciamento de informações. O método possui um processo bem-definido, cobrindo a análise e o projeto. A fase de análise se baseia no desenvolvimento de uma forma estendida do modelo entidade-relacionamento, denominada modelo OOA (*Obect-Oriented Analysis*). As máquinas de estados são utilizadas como modelos auxiliares na definição dos comportamentos de classes. O modelo OOA é utilizado como entrada da fase de projeto, onde será refinado. O método também fornece um conjunto de critérios para a avaliação dos projetos. (Coleman et al., 1996)

No método OMT o processo é dividido em três fases: análise, a qual se preocupa com a modelagem do mundo real; projeto, onde são tomadas as decisões a respeito dos subsistemas e aspectos gerais da arquitetura; e implementação, onde o projeto é codificado em uma linguagem de programação. (Coleman et al., 1996)

O método Booch reconhece que a análise e o projeto não podem prosseguir de forma mutuamente isolada, defendendo uma abordagem gradual; o projeto deve ser melhorado até que fiquemos satisfeitos. Booch denomina isso "*round-trip gestalt design*" (o que traduzido livremente corresponde a "teoria da forma com ida e volta"), sugerindo que esta estratégia é a base para o desenvolvimento orientado a objetos. Assim, o processo Booch é descritivo (dizendo o que *podemos* fazer) em vez de prescritivo (dizendo o que *devemos* fazer). (Coleman et al., 1996)

O método Objectory se baseia no conceito de **caso de utilização**. Um caso de utilização representa um diálogo (isso é, uma seqüência de transações) entre o sistema e um usuário, realizado para alcançar algum objetivo. Como outros

sistemas também podem ser "usuários", o Objectory utiliza o termo **agente** para abranger todos os usuários do sistema, sejam eles mecânicos ou humanos. Os casos de utilização tentam capturar a funcionalidade do sistema pela representação do que os agentes deveriam ser capazes de fazer com ele. Todos os outros modelos são construídos com base nas considerações feitas para os casos de utilização; como conseqüência, esses casos proporcionam elos de ligação entre as fases do Objectory. (Coleman et al., 1996)

O CRC (*Class–Responsibility–Collaboration* ou Classe–Responsabilidade–Colaborador) é uma técnica exploratória, e não um método completo. O CRC foi projetado, originalmente, como uma maneira para ensinar os conceitos básicos do projeto orientado a objetos. A técnica CRC pode ser explorada em outros métodos (por exemplo, o método Booch). O CRC lida, primariamente, com a fase de projeto do desenvolvimento. O processo é antropomórfico e orienta o desenvolvimento fazendo com que as equipes assumam o papel dos objetos em cenários de projeto. As classes são registradas em cartões de referência. (Coleman et al., 1996)

Os Métodos Formais representam uma abordagem de engenharia de software baseada na aplicação matemática discreta (ou seja, teoria de conjuntos, lógica, etc.) à programação de computadores. (Coleman et al., 1996)

O método Fusion integra os aspectos mais positivos de diferentes métodos. Os principais aspectos que influenciaram o método Fusion são: modelo de objetos e processos do OMT, interação de objetos do CRC, pré-condições e pós-condições dos Métodos Formais e visibilidade do Booch. (Coleman et al., 1996)

7.13 MÉTODO OOAD

Os métodos de análise de requisitos de software orientados a objeto possibilitam que se modele um problema ao representar classes, objetos, atributos e operações como componentes de modelagem primordiais. O ponto de vista orientado a objeto combina classificação de objetos, herança dos atributos e comunicação de mensagens no contexto de uma notação de modelagem. Os

objetos modelam quase todos os aspectos identificáveis do domínio de problemas: entidades externas, coisas, unidades organizacionais, lugares e estruturas; todos podem ser representados como objetos. (Pressman, 1995)

O método de análise orientada a objeto (OOA – *Object-Oriented Analysis*) proporciona uma notação e um conjunto de heurísticas para a construção de um modelo. Estruturas, sujeitos, conexões de instâncias e caminhos de mensagens são usados para se construir uma especificação gráfica de um sistema baseado em computador. O objetivo primário da OOA é identificar classes a partir das quais objetos possam ser apresentados como instâncias. (Pressman, 1995)

O projeto orientado a objeto (OOD – *Object-Oriented Design*) cria um modelo do mundo real que pode ser realizado em software. Os objetos oferecem um mecanismo para representar o domínio de informação, enquanto as operações descrevem o processamento que é associado ao domínio de informação. As mensagens – um mecanismo de interface – constituem os meios pelos quais as operações são invocadas. (Pressman, 1995)

A característica única de análise/projeto orientado a objeto (OOAD) é que os objetos “sabem” quais operações podem ser aplicadas a eles. Esse conhecimento é conseguido combinando-se dados e abstrações procedimentais num único componente de programa (denominado objeto ou pacote). (Pressman, 1995)

A metodologia OOAD desenvolveu-se como resultado de uma nova classe de linguagens de programação orientadas a objeto tais como a Smaltalk, C++, Objective-C, CLU, Ada, Modula e outras. Consequentemente, as representações de projeto orientado a objeto têm mais propensão que outras à dependência de uma linguagem de programação. (Pressman, 1995)

A OOAD constitui-se de uma abordagem de três passos que requer que se declare o problema, se defina uma estratégia de solução informal e se formalize a estratégia ao identificar objetos e operações, especificar interfaces e oferecer detalhes de implementação para abstrações procedimentais e de dados. O papel do OOD é pegar as classes e objetos básicos definidos como parte da OOA e refiná-los com detalhes de projeto adicionais. Os projetos são representados usando-se uma dentre uma série de notações gráficas e uma linguagem de projeto de programa. O

OOD oferece-nos os meios para quebrarmos as “partições” entre dados e processo. Ao fazermos isso, a qualidade de software pode ser melhorada. (Pressman, 1995)

Uma série de diferentes métodos têm sido sugeridos para a programação orientada a objeto. Todos fazem uso dos mesmos conceitos fundamentais, mas cada um introduz sua própria notação, heurística e filosofia. Com o objetivo de mostrar objetivos, benefícios e conceitos dessa programação, é utilizada a abordagem de Coad e Yourdon. Não obstante outras abordagens sejam igualmente importantes, essa é bem mais fácil de ser apresentada como uma visão geral.

Os objetivos fundamentais do OOD, segundo Coad e Yourdon (1993) são: melhorar a produtividade, aumentar a qualidade e elevar a manutenibilidade.

7.13.1 Melhorar a Produtividade

O OOD focaliza o empenho na atividade principal de projeto de software. Como retorno desse investimento, menos tempo é necessário para teste e remoção de defeitos. Mas a melhora geral na produtividade durante o desenvolvimento de um sistema pode ser de modestos 20%, ou ainda menos do que isso, se não houver familiaridade com o OOD. (Coad e Yourdon, 1993)

Mas existe outra perspectiva: em lugar de melhorar apenas a produtividade do desenvolvimento, procura-se melhorar a produtividade ao longo de todo o ciclo de vida. A maior parte das organizações reconhecem que entre 75% e 80% do custo ocorre depois do sistema ter sido colocado em operação; isso também significa que muitos defeitos só são descobertos após a entrega e, ainda mais importante, a maior parte da funcionalidade do sistema é acrescentada a ele depois do início de sua operação. Assim, um método que enfatiza manutenção melhora a produtividade da empresa por permitir aos programadores de manutenção efetuarem modificações mais rapidamente. Isso pode não impressionar ao usuário final que deseja um novo sistema desenvolvido prontamente, mas a organização pode ser capaz de reduzir o número de pessoas designadas para o serviço de manutenção e

liberar algumas delas para um novo trabalho de desenvolvimento. (Coad e Yourdon, 1993)

O OOD também pode melhorar a produtividade por estabelecer um mecanismo prático de **reutilização** de Classes de um projeto para outro. Isso é implementado usualmente com uma "biblioteca de Classes" que contém hierarquias de classes e subclasses. Desse modo, o desenvolvimento de uma biblioteca de Classes torna-se essencial. Para o primeiro projeto de OOD – em que uma tal biblioteca não existe – há pouca oportunidade para a reutilização. Porém, após poucos anos de elaboração e aprimoramento de uma biblioteca de Classes, como resultado de dezenas de projetos, essa biblioteca de Classes pode proporcionar os sinais iniciais de melhora na produtividade. Os novos sistemas são vistos cada vez mais como extensões ou aperfeiçoamentos de Classes que já fazem parte da biblioteca. (Coad e Yourdon, 1993)

7.13.2 Aumentar a Qualidade

A enorme ênfase na produtividade tem obscurecido a necessidade de melhorar a **qualidade** do software. A ausência de garantias para o software baseado em computador é um exemplo da inabilidade profissional para criar produtos com nível reconhecido de qualidade. Mas a qualidade é ignorada, conforme os sistemas tornam-se mais complexos e as conseqüências de falhas no software se tornam mais sérias. (Coad e Yourdon, 1993)

Existem muitas ferramentas, técnicas e métodos para melhorar a qualidade do software. Entretanto, várias organizações contam com *slogans* ou testes meticulosos do produto de software ao final do processo de desenvolvimento, em lugar de concentrar atenção no processo. Os processos que elaboram produtos de alta qualidade logo no início do desenvolvimento – especialmente a análise e o projeto – podem reduzir drasticamente a quantidade dos erros descobertos nas

últimas etapas do desenvolvimento, e podem melhorar grandemente a qualidade do sistema. (Coad e Yourdon, 1993)

Ao longo do tempo a percepção da qualidade do software assume um novo significado, especialmente à proporção que a indústria de software se torne mais competitiva. Conforme os defeitos em um sistema de software são reduzidos a um certo limiar, os usuários finais passam a considerar a qualidade como mais do que simplesmente a ausência de defeitos. A qualidade do software – adequação ao uso – começa a incluir a facilidade de utilização, a portabilidade e a capacidade de um sistema para lidar com mudanças contínuas. Esses aspectos tornam-se importantes para a aferição da qualidade do software. (Coad e Yourdon, 1993)

7.13.3 Elevar a Manutenibilidade

Os requisitos para um sistema estão sempre em permanentemente mudança devido a: clientes, competição, reguladores, demonstradores e especialistas. (Coad e Yourdon, 1993)

Um projetista empenha-se em organizar um projeto assim que ele se mostra resiliente o bastante a alterações; procura um empacotamento que permanecerá estável ao longo do tempo. Mas **como** o projetista pode organizar um projeto de forma a acomodar alterações que não podem ser antecipadas por anos ou décadas? A resposta é separar as partes do sistema que são intrinsecamente voláteis daquelas partes que aparentam ser mais estáveis. (Coad e Yourdon, 1993)

Qual será o impacto de inclusões, extensões, alterações e eliminações de recursos sobre um projeto de sistema? Isso é especialmente importante quando consideramos “famílias” de sistemas – isto é, situações nas quais uma variedade de implementações pode ser necessária. A estabilidade também é um fator importante por todo o ciclo de desenvolvimento de um sistema. (Coad e Yourdon, 1993)

Independentemente do método, um projeto de software deve dedicar-se ao processamento. Em uma abordagem baseada em objetos, os Serviços serão

necessários para criar um Objeto, conectar um Objeto a outros Objetos, calcular um resultado e proporcionar a monitoração de atividades. O grau de sofisticação dos Serviços é bastante instável; tal sofisticação está continuamente sujeita a quatro restrições – capacidade, planejamento, orçamento e pessoas. (Coad e Yourdon, 1993)

O projetista não evita a especificação de Serviços nem pode projetar de modo a evitar um Serviço necessário. Esse trabalho sempre tem de ser feito. A consideração importante é: o que vem primeiro? O que predomina? O que permanece estável? (Coad e Yourdon, 1993)

As interfaces externas são os próximos componentes de alteração mais provável. O quanto os dispositivos são inteligentes? O que os outros sistemas precisam receber do sistema sob consideração? O que esse sistema necessita receber dos outros? Quais serão as solicitações das pessoas quando estiverem utilizando o sistema? (Coad e Yourdon, 1993)

A próxima parte passível de mudanças inclui os Atributos que descrevem itens do domínio do problema. Ainda que essas alterações se apliquem a uma única Classe, por exemplo, “Avião” e seus Atributos. (Coad e Yourdon, 1993)

Os aspectos mais estáveis são as Classes que descrevem estritamente o domínio do problema e as responsabilidades do sistema dentro desse domínio. Por exemplo, quer se especifique um controle de estoque muito simples ou um sofisticado sistema de controle de tráfego aéreo, ainda assim ter-se-á as mesmas Classes básicas com as quais será organizada a análise e finalmente a especificação: “Avião”, “Controlador”, “Espaço Aéreo”, etc. Um sistema mais custoso terá mais Atributos para certas Classes e terá interfaces mais elaboradas para a monitoração de dispositivos e de outros sistemas (e Classes adicionais para modelar isso). O sistema mais caro terá Serviços mais sofisticados definidos para cada Classe (p. ex., “Avião” teria um Serviço de rastreamento automatizado, “Avião.Rastreamento”). Da mesma forma, a versão mais dispendiosa poderia ter algumas Classes adicionais (p. ex., “Radar” com os seus Atributos correspondentes, além de Serviços como “Radar.Examinar”). Por maior que seja o sistema, seu aspecto mais estável (as Classes no domínio do problema) permanecerá o mesmo

através de quaisquer alterações potencialmente importantes no escopo das responsabilidades do sistema. (Coad e Yourdon, 1993)

Ainda assim, o projetista prevê e planeja alterações e aceita cada modificação como um fato consumado, em vez de condená-la como produto de um julgamento descuidado. (Coad e Yourdon, 1993).

As motivações e benefícios da utilização do OOD, segundo Coad e Yourdon (1993) são:

a) Atacar os mais desafiadores domínios de problemas. A OOA traz grande ênfase à compreensão dos domínios de problemas. O OOD preserva a semântica do domínio do problema;
b) Aperfeiçoar a interação entre os especialistas em domínio do problema, analistas, projetistas e programadores. O OOD organiza os projetos utilizando os métodos de organização inerentes ao pensamento humano;
c) Aumentar a consistência interna durante a análise, o projeto e a programação. O OOD reduz a faixa de separação entre as diferentes atividades por tratar Atributos e Serviços como um todo intrínseco;
d) Representar explicitamente elementos comuns. O OOD usa a herança para identificar e capitalizar a presença de elementos comuns de Atributos e Serviços;
e) Construir sistemas resilientes a alterações. O OOD acondiciona as partes voláteis no interior de construções do domínio do problema, proporcionando estabilidade durante as alterações de requisitos ou de sistemas similares;
f) Reutilizar os resultados da OOA e do OOD – acomodando famílias de sistemas e os relacionamentos práticos no interior de um sistema. O OOD organiza os resultados com base em construções do domínio do problema de do domínio da implementação, para uma primeira reutilização e para as subseqüentes;
g) Fornecer uma representação básica consistente para OOA (o que está para ser construído) e para o OOD (como isso é para ser feito dessa vez). O OOD suporta uma continuidade de representação, para a expansão sistemática dos resultados da OOA para o OOD.

7.14 CONCEITOS E PRINCÍPIOS FUNDAMENTAIS

Primeiramente, são apresentados conforme Coad e Yourdon (1993) os conceitos de objeto e classe:

Do dicionário:

Objeto: [algo deixado pelo caminho (Latim Medieval); uma disposição anterior (Latim)] Uma pessoa ou coisa para a qual a ação, o pensamento ou o sentimento é dirigido. Qualquer coisa visível ou tangível; uma substância ou produto material.

Classe: [divisão em castas do povo Romano (Latim); uma chamada ou citação (Grego)] Uma quantidade de pessoas ou coisas agrupadas devido a semelhança de traços comuns. [Webster's, 1977]

E, mais especificamente, para o OOAD:

Objeto: Uma **abstração** de alguma coisa no domínio do problema ou em sua implementação, refletindo a capacidade de um sistema para manter informações sobre ela, interagir com ela, ou ambos: um **encapsulamento** de valores de Atributo e seus Serviços exclusivos. (Sinônimos: uma Instância ou uma Ocorrência)

Classe: Uma descrição de um ou mais Objetos, através de um conjunto uniforme de Atributos e Serviços; além disso, pode conter uma descrição de como criar novos Objetos na Classe.

A seguir, os termos análise e projeto conforme Coad e Yourdon (1993):

Do dicionário:

Análise: [dissolução, uma decomposição do todo em partes (Grego)] Uma separação ou decomposição de um todo em suas partes a fim de descobrir sua natureza, proporção, função, relacionamento, etc.

Projeto: [um planejamento (Latim)] A elaboração de planos, esboços, padrões, etc. originais. [adaptado de Webster's, 1977]

E, mais especificamente:

Análise: O ato de estudar um domínio de problema, conduzindo à especificação de comportamentos externamente observáveis; uma declaração completa, consistente e exeqüível, daquilo que é necessário; uma relação de características, tanto funcionais como operacionais quantificadas (p. ex., confiabilidade, disponibilidade, performance).

Projeto: O ato de tornar uma especificação de comportamentos externamente observáveis e acrescentar-lhe os detalhes requeridos pela implementação real de um sistema de computação, incluindo os detalhes de interação humana, gerenciamento de tarefas e gerenciamento de dados.

De acordo com Coad e Yourdon (1993), os conceitos de abstração, abstração de dados e encapsulamento são:

Do dicionário:

Abstração: O princípio de ignorar os aspectos de um assunto que não sejam relevantes para o propósito em questão, a fim de dedicar uma concentração maior aos aspectos relevantes. [*Oxford*, 1986]

Quando alguém utiliza a abstração, está admitindo que um artefato ou porção do mundo real é algo complexo; em lugar de tentar compreender o todo, seleciona apenas parte desse todo.

A abstração de procedimentos é freqüentemente caracterizada como uma abstração de "função/subfunção". A divisão de uma seqüência de processamento em subpassos é um método básico de tratamento da complexidade. Porém, utilizar tal decomposição na organização de um projeto tem qualquer coisa de arbitrário e altamente volátil. Um mecanismo de abstração mais eficiente é a abstração de dados. Esse princípio pode ser uma base para a organização do pensamento e especificação das responsabilidades de um sistema.

Do dicionário:

Abstração de dados: Consiste na definição de um tipo de dados de acordo com as operações aplicáveis aos objetos desse tipo, com a restrição de que os valores de tais objetivos só podem ser modificados e observados pelo uso daquelas operações.

[*Oxford,* 1986]

Na aplicação da abstração de dados, pode-se definir os Atributos e os Serviços que manipulam exclusivamente esses Atributos. A única forma de chegar aos Atributos é através de um Serviço. Os Atributos e seus Serviços podem ser tratados como um todo intrínseco.

Do dicionário:

Encapsulamento: (Ocultação da Informação) Um princípio, usado no desenvolvimento de uma estrutura global de programa, segundo o qual cada componente de um programa deve encapsular ou ocultar uma só decisão de projeto.... A interface para cada módulo é definida de modo a revelar o mínimo possível sobre o seu funcionamento interno. [*Oxford*, 1986]

O encapsulamento ajuda a minimizar o trabalho no desenvolvimento de um novo sistema. O encapsulamento mantém juntos os aspectos relacionados, minimiza as influências mútuas entre as diferentes partes do trabalho e isola certos requerimentos específicos de outras partes da especificação que poderiam utilizá-los.

Observe que a abstração de dados é uma forma dessa característica de “manter juntos os aspectos relacionados” que o encapsulamento apresenta. Observe também que a comunicação com mensagens (descrita mais adiante neste capítulo) é uma forma da característica de “interface restrita” do encapsulamento.

Um outro conceito importante, ainda segundo Coad e Yourdon (1993), é a herança (representando generalização – especialização):

Herança: Um mecanismo para expressar a similaridade entre Classes, simplificando a definição de Classes semelhantes a outras que foram definidas anteriormente. Ela representa a generalização e a especialização, tornando explícitos Atributos e Serviços comuns em uma herança ou ordenação de Classe.

Esse conceito forma a base para uma técnica significativa de representação explícita de elementos comuns. A herança permite especificar Atributos e Serviços comuns apenas uma vez, bem como especializar e estender esses Atributos e Serviços em casos específicos. A herança pode ser aplicada para representar explicitamente aspectos comuns, a partir das primeiras atividades de análise e continuando até o projeto.

E, para finalizar os conceitos associação e comunicação de mensagens de Coad e Yourdon (1993):

Do dicionário:

Associação: A união ou conexão de idéias. [Webster’s, 1977]

As pessoas utilizam a associação para agrupar certas coisas que acontecem em um tempo determinado ou sob circunstâncias semelhantes; por exemplo, unindo um veículo e um proprietário, um escrivão e um evento legal.

Do dicionário:

Mensagem: Qualquer comunicação, escrita ou oral, enviada entre pessoas. [Webster’s, 1977]

A comunicação com mensagens é um conceito especificamente para as interfaces. Ele está relacionado ao encapsulamento, tendo em vista que os detalhes da ação a ser realizada são encapsulados dentro do receptor de uma mensagem.

LINGUAGENS DE PROGRAMAÇÃO

A evolução das principais Linguagens de Programação é descrita neste item, o qual segue cronologicamente o desenvolvimento de algumas delas, concentrando-se nas contribuições da linguagem e na motivação para seu desenvolvimento. Em geral, de especial interesse são os recursos que mais influenciaram as linguagens. E, mais especificamente as influências sobre a linguagem de programação abordada pelo presente estudo, o Visual Basic.

A figura 1 é um gráfico da genealogia das linguagens de alto nível.

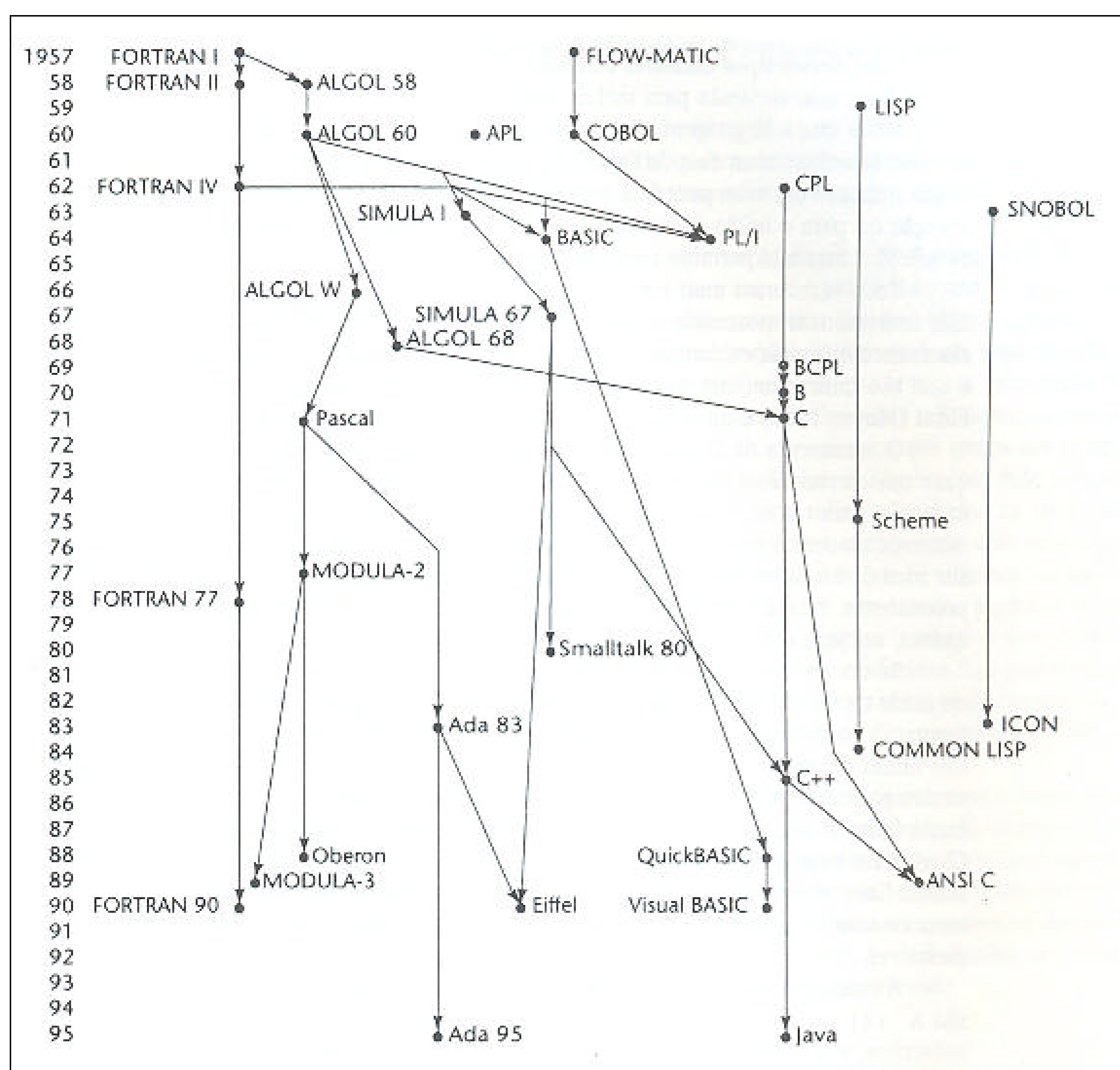

Figura 1: Genealogia das linguagens de programação
Fonte: Sebesta (2000)

A primeira linguagem discutida é bastante incomum sob vários aspectos. Por um lado ela nunca foi implementada. Além disso, não obstante haver sido desenvolvida em 1945, sua descrição não foi publicada até 1972. Em conseqüência da ignorância geral a respeito da linguagem, algumas de suas capacidades não apareceram em outras linguagem até 15 anos depois do seu desenvolvimento. A linguagem Plankalkül, que significa cálculo de programa, foi desenvolvida pelo alemão Konrad Zuse entre 1936 e 1945. Essa linguagem era notavelmente completa, com alguns dos recursos mais avançados na área das estruturas de dados. (Sebesta, 2000)

Os computadores que se tornaram disponíveis no final da década de 40 e no início da década de 50 eram bem menos usáveis do que os de hoje. Além de serem lentos, pouco confiáveis e caros, e possuírem memórias extremamente pequenas, eram difíceis de programar por causa da falta de software de apoio. Não havia nenhuma linguagem de programação de alto nível ou mesmo linguagens *assembly*, de forma que a programação era feita em código de máquina, o que, além e tedioso, está propenso a erros. (Sebesta, 2000)

Esses problemas ocorrem com todas as linguagens de máquina e foram as principais motivações para inventar as linguagens *assembly*. Além disso, a maioria dos problemas de programação daquela época eram numéricos e exigiam operações aritméticas com ponto-flutuante e indexação para permitir o uso conveniente de *arrays*. (Sebesta, 2000)

Nenhuma dessas capacidades, entretanto, foi incluída na arquitetura dos computadores do final da década de 1940 e do início da década de 1950. As deficiências naturalmente levaram ao desenvolvimento de linguagens de nível bem mais elevado. (Sebesta, 2000)

A primeira dessas novas linguagens foi chamada de *Short Code* e desenvolvida em 1949. A *Short Code* não era traduzida para código de máquina; ao contrário, era implementada com um interpretador puro. Naquela época, esse processo era chamado de programação automática. Evidentemente, ele simplificava o processo de programação, mas à custa do tempo de execução. A interpretação da

Short Code era, aproximadamente, 50 vezes mais longa do que o código de máquina. (Sebesta, 2000)

Também é importante mencionar que as linguagens *assembly* desenvolveram-se durante o início da década de 1950. Porém, elas tiveram pouco impacto sobre o projeto de linguagens de alto nível. (Sebesta, 2000)

Certamente um dos maiores avanços particulares na computação veio com a introdução do computador IBM 704 em 1954, em grande medida porque suas capacidades motivaram o desenvolvimento do FORTRAN. Pode-se argumentar que, se não tivesse sido a IBM com o 704 e o FORTRAN, logo surgiria alguma outra organização com um computador similar e com uma linguagem de alto nível relacionada. (Sebesta, 2000)

Uma das principais razões pelas quais se tolerou os sistemas interpretativos desde o final da década de 1940 até meados da década de 1950 foi a falta de hardware de ponto-flutuante nos computadores disponíveis. Todas as operações com ponto-flutuante tinham de ser simuladas em software, o que consumia muito tempo. O anúncio do sistema IBM 704, possuindo tanto instruções de indexação como de ponto-flutuante em hardware, marcou o fim da era interpretativa. Apesar dos vários trabalhos desenvolvidos anteriormente, a primeira linguagem de alto nível compilada bem aceita foi o FORTRAN. (Sebesta, 2000)

Mesmo antes do sistema 704 ter sido anunciado em meados de 1954, os planos para o FORTRAN já haviam sido iniciados. No final deste mesmo ano, foi produzido o relatório que descrevia a versão inicial. A linguagem implementada, chamada FORTRAN I, é descrita no primeiro manual de referência da linguagem publicado em 1956. (Sebesta, 2000)

O sucesso inicial do FORTRAN é mostrado pelos resultados de uma pesquisa feita em abril de 1958. Naquela época, quase metade do código escrito para os 704 ainda era feita em FORTRAN, apesar do extremo ceticismo da maior parte do mundo da programação apenas um ano antes. (Sebesta, 2000)

O compilador FORTRAN II foi distribuído na primavera de 1958. Ele resolveu muitos dos problemas existentes no sistema de compilação FORTRAN I e acrescentou alguns recursos importantes à linguagem, sendo o mais destacado a compilação independente de sub-rotinas. O FORTRAN III foi desenvolvido, mas

jamais distribuído popularmente. O FORTRAN IV, porém, tornou-se uma das linguagens de programação mais usadas de seu tempo. Ela evoluiu ao longo do período de 1960 a 1962 e foi a versão padrão até 1978. (Sebesta, 2000)

O ALGOL 60 passou a existir como resultado dos esforços para projetar uma linguagem universal. Na década de 1950, entidades de vários países formaram comitês de estudo voltados para à criação de uma linguagem de programação universal, em resposta a proliferação de novas linguagens que tornou difícil a comunicação entre os usuários. Em meados de 1958, como resultado dos trabalhos de um encontro realizado na cidade de Zurique foi projetada a linguagem ALGOL 58. De muitas maneiras, o ALGOL 58 descendeu do FORTRAN, o que é muito natural. Ele generalizou muitos dos recursos deste e adicionou diversas construções e conceitos novos. Algumas das generalizações tinham a ver com a meta de não ligar a linguagem a qualquer máquina particular, e outras tentativas de tornar a linguagem mais flexível e poderosa. Uma combinação rara de simplicidade e de elegância surgiu desse esforço. (Sebesta, 2000)

No início de 1960, realizou-se o segundo encontro sobre o ALGOL, em Paris. O relatório ALGOL 60 foi publicado em meados de 1960. Sob alguns aspectos, o ALGOL 60 foi um sucesso; sob outros, um obscuro fracasso. Toda linguagem de programação imperativa projetada desde 1960 deve alguma coisa ao ALGOL. De fato, a maioria é uma descendente direta ou indireta. No entanto, o ALGOL 60 jamais conseguiu uso generalizado ou até mesmo significativo nos Estados Unidos ou na Europa. Há uma série de motivos para a falta de aceitação. Alguns dos recursos ficaram flexíveis demais, eles tornavam o entendimento difícil e a implementação ineficiente. Não obstante houvesse muitos outros problemas, o entrincheiramento do FORTRAN entre os usuários e a falta de suporte por parte da IBM provavelmente foram os fatores mais importantes para que o ALGOL 60 não tenha disseminado seu uso. (Sebesta, 2000)

O BASIC (*Beginners All-purpose Symbolic Instruction Code*) é outra linguagem de programação que desfrutou de um uso generalizado, mas obteve pouco respeito. Ele foi muito ignorado pelos cientistas da computação. Além disso, em suas primeiras versões, o BASIC era deselegante e incluía somente um escasso conjunto de instruções de controle. Ele foi muito popular nos microcomputadores no

final da década de 1970 e início de 1980. No início da década de 1990, houve um ressurgimento no uso do BASIC com o lançamento do Visual BASIC pela Microsoft em 1991. (Sebesta, 2000)

O aspecto mais importante do BASIC original é que ele foi o primeiro método amplamente usado de acesso por terminal remoto a um comutador. Os terminais haviam começado a tornar-se disponíveis naquela época. Antes disso, a maioria dos programas era produzida nos computadores ou por meio de cartões perfurados ou por fita de papel. (Sebesta, 2000)

Grande parte do projeto do BASIC veio do FORTRAN, com uma pequena influência da sintaxe do ALGOL 60. Posteriormente, ele cresceu de muitas maneiras, com pouco ou nenhum esforço para padronizá-lo. O *American National Standards Institute* publicou um padrão *Minimal BASIC* (ANSI, 1978), mas esse representava somente o mínimo de recursos da linguagem. De fato, o BASIC original era muito semelhante à *Minimal BASIC*. (Sebesta, 2000)

Obviamente, as primeiras versões da linguagem não se destinavam e não deviam ser usadas para programas sérios de qualquer tamanho significativo. As versões posteriores são muito melhor adequadas a essas tarefas. As razões mais prováveis para o sucesso do BASIC são a facilidade com que ele pode ser aprendido e a sua facilidade de implementação, mesmo em computadores muito pequenos. (Sebesta, 2000)

Duas das versões contemporâneas do BASIC, agora amplamente usadas, são o QuickBASIC e o Visual BASIC. Ambas rodam em microcomputadores. O Visual Basic baseia-se no QuickBasic, mas foi projetado para desenvolver sistemas de software que têm interfaces com o usuário providas de janelas. O Visual BASIC também tem sido usado como uma linguagem de *scripting* para programação CGI (*Common Gateway Interface*). (Sebesta, 2000)

A linhagem do BASIC é mostrada na Figura 2.

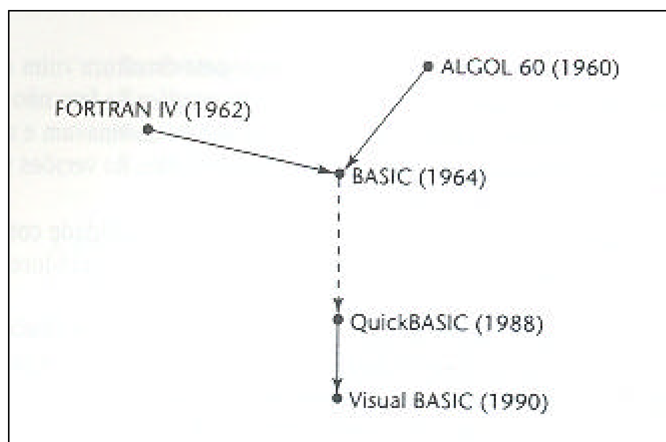

Figura 2: Genealogia do BASIC
Fonte: Sebesta (2000)

7.15 LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETO

As linguagens que suportam programação orientada a objeto agora estão firmemente em uma posição de destaque, foram criados dialetos que suportam programação orientada a objeto e incluídos em praticamente todas as linguagens de programação. Entre elas estão o C++, a Ada 95 e a CLOS, uma versão orientada a objeto do LISP. O C++ e a Ada 95 suportam programação orientada a procedimentos e a dados, além da programação orientada a objeto. A CLOS também suporta programação funcional. Algumas das linguagens mais recentes projetadas para suportar a programação orientada a objeto não suportam outros paradigmas, mas ainda empregam algumas das estruturas básicas e têm aparência das linguagens imperativas mais antigas. Entre elas estão a Eiffel e o Java. Por fim, há uma linguagem puramente orientada a objeto bastante não convencional: a Smalltalk. Ela foi a primeira a oferecer suporte completo para programação orientada a objeto. O suporte específico à programação orientada a objeto varia amplamente entre as linguagens. (Sebesta, 2000)

A programação orientada a objeto é essencialmente uma aplicação do princípio da abstração de tipos de dados. Especificamente, na programação orientada a objeto, os atributos comuns de uma coleção de tipos de dados abstratos

similares são fatorados e colocados em um novo tipo. Os membros da coleção herdam as partes comuns deste. Isso é herança, a qual é o centro da programação orientada a objeto e das linguagens que a suportam. (Sebesta, 2000)

7.15.1 Programação Orientada a Objeto

O conceito de **programação orientada a objeto** tem suas raízes na linguagem de programação SIMULA 67, mas não foi amplamente desenvolvido até que a evolução da Smalltalk resultasse na produção da versão de 1980. De fato, alguns consideram a Smalltalk a única linguagem puramente orientada a objeto. Uma linguagem com esta característica deve oferecer três recursos chaves: tipos de dados abstratos, herança e um tipo particular de vinculação dinâmica. (Sebesta, 2000)

A programação orientada por procedimentos, o paradigma de desenvolvimento de objeto mais popular na década de 1970, concentra-se em subprogramas e em bibliotecas de subprogramas. Dados são enviados a subprogramas para computações. Por exemplo, um *array* (estrutura ordenada de elementos acessíveis individualmente) de valores inteiros que precisa ser classificado é enviado como um parâmetro a um subprograma que o classifica. (Sebesta, 2000)

A **programação orientada a dados** concentra-se em tipos de dados abstratos. Nesse paradigma, a computação de um objeto-dados é especificada chamando-se subprogramas associados com o objeto-dados. Se um objeto *array* precisar ser classificado, a operação será definida no tipo de dados abstrato para o *array*. O processo de classificação é ativado chamando-se essa operação no objeto *array* específico. O paradigma da programação orientada a dados foi popular na década de 1980, e é bem servida pelas facilidades de abstração de dados do Modula-2, da Ada e de diversas linguagens mais recentes. As linguagens que

suportam programação orientada a dados freqüentemente são conhecidas como baseadas em dados. (Sebesta, 2000)

7.15.2 Conceitos

7.15.2.1 Abstração

Uma abstração é uma visualização ou uma representação de uma entidade que inclui somente os atributos de importância em um contexto particular. Ela permite que se colete instâncias de entidades em grupos cujos atributos comuns das mesmas não precisam ser considerados. Estes atributos comuns são abstraídos. Dentro dos grupos, somente os atributos que distinguem os elementos individuais precisam ser considerados. Isso resulta em uma significativa simplificação dos elementos do grupo. Visualizações menos abstratas dessas entidades devem ser consideradas quando é necessário ver um nível mais elevado de detalhe. A abstração é um recurso contra a complexidade da programação, seu propósito é simplificar o processo de programação. Ela é um recurso eficiente porque permite que os programadores concentrem-se nos atributos essenciais e ignorem os atributos subordinados. (Sebesta, 2000)

Os dois tipos fundamentais de abstração nas linguagens de programação contemporâneas são a de processo e a de dados. (Sebesta, 2000)

O conceito de **abstração de processo** está entre os mais antigos no projeto de linguagens de programação. Todos os subprogramas são abstrações de processo porque oferecem uma maneira do programa especificar que algum processo deve ser feito, sem oferecer os detalhes de como será feito (pelo menos, no programa que faz a chamada). Por exemplo, quando um programa precisa classificar uma matriz de objetos de dados numéricos de algum tipo, normalmente

usa um subprograma para o processo de classificação. No ponto em que o processo de classificação é necessário, uma instrução como

```
sort_int(list, list_len)
```

é colocada no programa. Essa chamada é uma abstração do processo de classificação real, cujo algoritmo não é especificado. A chamada é independente do algoritmo implementado no subprograma chamado. (Sebesta, 2000)

No caso do programa `sort_int`, os únicos atributos essenciais são o nome da matriz a ser classificada, o tipo de seus elementos, seu tamanho e o fato de que a chamada a `sort_int` resultará nela ser classificada. O algoritmo particular que `sort_int` implementa é um atributo não essencial para o usuário. (Sebesta, 2000)

A abstração de processo é crucial para a programação. A capacidade de abstrair muitos dos detalhes dos algoritmos em subprogramas torna possível construir, ler e entender programas grandes. Lembre-se de que para ser considerado um programa grande agora, ele deve ter pelo menos várias centenas de milhares de linhas de código. (Sebesta, 2000)

Todos os subprogramas, inclusive concorrentes e os manipuladores de exceções, são abstrações de processo. (Sebesta, 2000)

A evolução da abstração de dados seguiu necessariamente à da abstração de processo, porque uma parte integrante e central de toda abstração de dados são suas operações definidas como abstrações de processo. (Sebesta, 2000)

7.15.2.2 Encapsulamento

Preliminarmente à apresentação dos tipos de dados abstratos, devemos discutir o encapsulamento, um precursor e um mecanismo de suporte para estes. (Sebesta, 2000)

Quando o tamanho do programa estende-se para além de alguns milhares de linhas, dois problemas práticos aparecem. Do ponto de vista do programador, fazer com que esse programa apareça como uma única coleção de subprogramas não impõe um nível de organização adequado ao programa para mantê-lo intelectualmente administrável. Uma solução é organizá-lo em recipientes (*containers*) sintáticos que incluem grupos de subprogramas e de dados logicamente relacionados. Esses recipientes sintáticos, muitas vezes são chamados **módulos**, e o processo de projetá-los é chamado **modularização**. O segundo problema prático em relação a programas maiores é a recompilação. No caso de um pequeno, recompilá-los em sua extensão depois de cada modificação não é custoso. Mas quando os programas vão além de alguns milhares de linhas, o custo da recompilação deixa de ser insignificante. Assim, há uma necessidade evidente de encontrar maneiras de evitar a recompilação das partes de um programa não afetadas por uma mudança. Isso pode ser obtido organizando-se os programas em coleções de subprogramas e de dados, cada um dos quais pode ser compilado sem a recompilação do restante do programa. Essa coleção é chamada de **unidade de compilação**. (Sebesta, 2000)

O encapsulamento é um agrupamento de subprogramas e de dados que eles manipulam. O encapsulamento, separada ou independentemente compilável, constitui um sistema abstraído e uma organização lógica para uma coleção de computações relacionadas. Portanto, o encapsulamento resolve ambos os problemas práticos descritos acima. (Sebesta, 2000)

Os encapsulamentos, muitas vezes, são colocados em bibliotecas e postos à disposição para serem reutilizados em programas que não para os quais eles foram escritos. (Sebesta, 2000)

Pessoas têm escrito programas com mais de alguns milhares de linhas ao longo dos últimos 40 anos, de modo que as técnicas para oferecer encapsulamento vem sendo desenvolvidos há algum tempo. (Sebesta, 2000)

Em muitas das linguagens assemelhadas ao ALGOL, os programas podem ser organizados aninhando-se definições de subprogramas dentro de subprogramas logicamente maiores que os usam. Tal método de organizar programas, que usa escopo estático, está longe de ser o ideal. Além disso, em algumas linguagens os

subprogramas não são unidades de compilação. Portanto, eles não criam boas construções de encapsulamento. (Sebesta, 2000)

No FORTRAN 77, subprogramas podem ser coletados em arquivos, compilados independentemente e colocados em bibliotecas. Coleções de definições de blocos `COMMON` também podem ser manipuladas dessa maneira. Essa é uma técnica de organização eficiente, mas não é feita nenhuma verificação de interface quando tais encapsulamentos são usados de maneira que essa abordagem é inerentemente insegura. (Sebesta, 2000)

Na linguagem C, uma coleção de funções e de definições de dados relacionadas pode ser colocada em um arquivo, que pode ser, então, compilado independentemente. Não obstante os sistemas de compilação C agora verificarem a justeza das interfaces de funções apropriadamente definidas, eles ainda não verificam o tipo de definições de dados de diferentes arquivos. Assim, os arquivos C também não criam encapsulamentos seguros. (Sebesta, 2000)

Muitas linguagens contemporâneas, inclusive o FORTRAN 90 e a Ada, oferecem a capacidade de reunir coleções de subprogramas, de tipos e de dados em unidades que podem ser compiladas separadamente, significando que suas informações de interface são salvas pelo compilador e usadas para verificação de tipo de interface quando usadas por outra unidade. Essas linguagens também incluem mecanismos de controle de acesso para as entidades nessas unidades. Isso permite que a unidade tenha alguns nomes de tipo visíveis a unidades externas, enquanto tem a representação desses tipos visível somente a outras entidades da unidade. Estas últimas fazem encapsulamentos perfeitos. Elas não somente suportam uma organização de programa concisa e lógica, mas também tornam-na evidente para os leitores do programa. (Sebesta, 2000)

7.15.2.3 Abstração de Dados

Um tipo de dado abstrato, colocando de maneira simples, é um encapsulamento que inclui somente a representação de dados de um tipo específico de dado e os subprogramas que fornecem as operações para esse tipo. Por meio do controle de acesso, detalhes desnecessários do tipo podem ser ocultos das unidades fora do encapsulamento que o usam. As unidades de programa que usam dados abstratos podem declarar variáveis desse tipo, não obstante a representação real estar oculta delas. Uma instância de um tipo de dado abstrato é chamada de **objeto**. (Sebesta, 2000)

Uma das motivações para a abstração de dados é similar à da abstração de processo. Ela é um recurso contra a complexidade, um meio de tornar programas grandes e/ou complicados mais manejáveis. Da mesma forma que a presença da abstração de processo permite uma metodologia de projeto de programa diferente, a disponibilidade de abstração de dados também a permite. (Sebesta, 2000)

A programação orientada a objetos é um resultado do uso da abstração de dados no desenvolvimento de software, e um de seu mais importantes componentes. (Sebesta, 2000)

7.15.2.3.1 Ponto-flutuante como um Tipo de Dado Abstrato

O conceito de tipos de dados abstratos, pelo menos em termos de tipos incorporados, não é um desenvolvimento recente. Todos eles, até mesmo os do FORTRAN I, são tipos de dados abstratos, não obstante raramente serem chamados assim. Por exemplo, considere um tipo ponto-flutuante. A maioria das linguagens inclui pelo menos uma implementação desse tipo, o que constitui um meio de criar variáveis para dados de ponto-flutuante e também fornece um conjunto de operações aritméticas para manipular objetos do tipo. (Sebesta, 2000)

Os tipos ponto-flutuante em linguagens de alto nível empregam um conceito chave da abstração de dados: ocultação de informação. O formato real do valor de dados em uma célula de memória de ponto-flutuante é oculto do usuário. As únicas operações disponíveis dão aquelas oferecidas pela linguagem. O usuário não tem permissão para criar novas operações sobre os dados do tipo, exceto aquelas que podem ser construídas usando as operações incorporadas. O usuário não pode manipular diretamente as partes da representação real de objetos de ponto-flutuante porque essa representação está oculta. É esse recurso que permite a portabilidade do programa entre implementações de uma linguagem particular, não obstante elas poderem usar diferentes representações de valores de poto-flutuante. (Sebesta, 2000)

7.15.2.3.2 Tipos de Dados Abstratos Definidos pelo Usuário

O conceito de tipos de dados abstratos definidos pelo usuário é relativamente recente. Um tipo de dado abstrato definido pelo usuário apresenta as mesmas características oferecidas pelos tipos ponto-flutuante: uma definição de tipo que permite que as unidades de programa declarem suas variáveis, mas oculta a sua representação, e um conjunto de operações para manipular objetos do tipo. (Sebesta, 2000)

Definimos agora formalmente um tipo de dado abstrato no contexto dos tipos definidos pelo usuário. Um **tipo de dado abstrato** é um tipo de dado que satisfaz as duas condições seguintes:

a) A representação ou a definição do tipo e as operações sobre objetos do tipo estão contidas em uma única unidade sintática. Além disso, outras unidades de programa podem ter permissão para criar variáveis do tipo definido;
b) A representação de objetos do tipo não é visível pelas unidades de programa que usam o tipo de modo que as únicas operações diretas

possíveis sobre esses objetos são aquelas oferecidas na definição do tipo. (Sebesta, 2000)

As unidades de programa que usam um tipo de dado abstrato específico são chamadas **clientes** desse tipo. (Sebesta, 2000)

As principais vantagens de empacotar a representação e as operações em uma única unidade sintática são as mesmas que as do encapsulamento. Constitui um método de organizar um programa em unidades lógicas que podem ser compiladas separadamente. Além disso, permite que modificações nas representações ou operações do tipo sejam feitas em uma única área do programa. Há diversas vantagens em ocultar detalhes da representação. A mais importante delas é os clientes não serem capazes de "ver" os detalhes da representação e, assim, seu código não depende dessa representação. Isso resulta em representações que podem ser modificadas a qualquer hora sem exigir mudanças nos clientes. A interface para a abstração representa alguns, mas não todos, os seus atributos. (Sebesta, 2000)

Outro benefício distinto e importante da ocultação de informação é o aumento da confiabilidade. Os clientes não podem mudar as representações subjacentes de objetos diretamente, seja intencional ou acidentalmente, aumentando assim a integridade desses objetos. Estes podem ser modificados somente pelas operações fornecidas. É difícil exagerar na importância de ocultar os detalhes de representação de um tipo de dado abstrato. (Sebesta, 2000)

7.15.2.3.3 Um Exemplo

Suponhamos que um tipo de dado abstrato deva ser construído para uma pilha que tem as seguintes operações abstratas:

create(pilha)	Cria e possivelmente inicializa um objeto da pilha;
destroy(pilha)	Desaloca o armazenamento da pilha;

empty(pilha) Uma função predicada (ou *boolean*) que retorna *true* (verdadeiro) se a pilha especificada estiver vazia, e *false* (falso) se não estiver;

push(pilha, elemento) Empurra o elemento especificado para a pilha especificada;

pop(pilha) Remove o elemento do topo da pilha especificada. (Sebesta, 2000)

Note que alguns projetos de implementação de tipos de dados abstratos não exigem as operações de criar e de destruir. Por exemplo, simplesmente definir uma variável para que seja de um tipo de dado abstrato pode criar implicitamente a estrutura de dados subjacente e inicilizá-la. (Sebesta, 2000)

Um cliente do tipo pilha poderia ter uma seqüência de código como a seguinte:

```
...
create(STK1);
push(STK1, COR1);
push(STK1, COR2);
if (not empty(STK1))
     then TEMP := top(STK1);
...
```

Suponhamos que a implementação original da abstração da pilha use uma representação de adjacência (que implemente pilhas em *arrays*). Em um momento posterior, devido a problemas de gerenciamento da memória com a representação de adjacências, ela é modificada para uma representação de lista ligada. Uma vez que foi usada a abstração de dados, tal mudança pode ser feita no código que define o tipo de pilha, mas nenhuma mudança será necessária em qualquer um dos clientes da abstração de pilha. Em especial, a seqüência de código acima não precisa ser modificada. Obviamente, uma mudança de protocolo de qualquer uma das operações exigiria mudanças nos clientes. (Sebesta, 2000)

Se a pilha não fosse implementada como um tipo de dado abstrato, essa mudança exigiria que os clientes do tipo pilha ficassem moldados à nova representação. Suponhamos, por exemplo, que as operações da pilha foram implementadas em Ada para operar sobre *arrays*. A mudança para representação de lista ligada exigiria que os clientes fossem modificados para enviar ponteiros em vez de nomes de *array* como parâmetros para os procedimentos de operação de pilha. Nesse caso, o protocolo de algumas operações *precisam* ser modificados, forçando, assim, mudanças nos clientes. (Sebesta, 2000)

Em suma, a meta da abstração de dados é permitir que programas definam tipos de dados com as características e com o comportamento de tipos incorporados. (Sebesta, 2000)

7.15.2.4 Herança

De meados até o final da década de 1980, tornou-se claro a muitos desenvolvedores de software que uma das melhores oportunidades para uma maior produtividade em suas profissões era a reutilização de software. Os tipos de dados abstratos, com seu encapsulamento e com seus controles de acesso, evidentemente eram as unidades a serem reutilizadas. O problema com a reutilização dos tipos de dados abstratos é que, em quase todos os casos, os recursos e as capacidades do tipo existente não são muito certos para o novo emprego. O antigo exige no mínimo algumas pequenas modificações que podem ser difíceis, porque exigem que a pessoa que faz a modificação entenda parte, se não todo, o código existente. Além disso, em muitos casos, as modificações exigem mudanças em todos os programa-clientes. (Sebesta, 2000)

Um segundo problema com a programação orientada a dados que todas as definições de tipos de dados abstratos são independentes e estão no mesmo nível. Isso, freqüentemente, torna impossível estruturar um problema para ajustar o seu espaço encaminhado pelo programa. Em muitos casos, o problema subjacente tem

categorias de objetos relacionadas, tanto como parentes (sendo similares uns aos outros) como pais e filhos (tendo algum tipo de relação subordinada). (Sebesta, 2000)

A herança oferece uma solução tanto para o problema da modificação apresentado pela reutilização de tipos de dados abstratos, como pelo problema de organização do programa. Se um novo tipo de dado abstrato puder herdar os dados e a funcionalidade de algum tipo existente, e se também for permitido modificar algumas dessas entidades e adicionar novas, a reutilização será grandemente facilitada sem exigir mudanças no tipo de dado abstrato reusado. Os programadores podem pegar um desses últimos existente e moldá-lo para que se ajuste à nova exigência do problema. Por exemplo, suponhamos que um programa já tenha um tipo de dado abstrato para *arrays* de inteiros que inclua uma operação de classificação. Depois de algum período de uso, o programa é atualizado e exige um tipo de dados abstrato para *arrays* de inteiros com a operação de classificação, mas também precisa de uma operação para comutar a mediana dos elementos dos objetos-*array*. Uma vez que a estrutura de *array* é oculta em um tipo de dado abstrato, sem herança, ele deve ser modificado para adicionar a nova operação nessa estrutura. Com herança, não há necessidade de modificar o tipo existente; pode-se definir uma subclasse dele que retenha as operações de classificação, mas adicione outra para a computação mediana. (Sebesta, 2000)

Os tipos de dados abstratos em linguagens orientadas a objeto, seguindo a linha da SIMULA 67, usualmente são chamados de classes. Como acontecem com as instâncias dos tipos de dados abstratos, as instâncias de classe são chamadas **objetos**. Uma classe definida pela herança de outra é uma **classe derivada** ou uma **subclasse**. Uma classe da qual a nova é derivada é a sua **classe-pai** ou **superclasse**. Os subprogramas que definem as operações em objetos de uma classe são chamados **métodos**. As chamadas a métodos freqüentemente são chamadas de **mensagens**. A coleção inteira de métodos de um objeto é chamada de **protocolo de mensagem** ou de **interface de mensagem** do objeto. Uma mensagem deve ter, no mínimo, duas partes: o objeto específico ao qual ela está sendo enviada e um nome de um método que defina a ação solicitada naquele.

Assim, as computações em um programa orientado a objeto são especificadas pelas mensagens enviadas de objetos a outro objetos. (Sebesta, 2000)

No caso mais simples, uma classe herda todas as entidades (variáveis e métodos) de sua classe-pai. Isso pode ser complicado pelos controles de acesso nas entidades de uma classe-pai. Nas definições de tipo de dados abstratos, algumas das entidades são classificadas como públicas e outras como privadas. Esses controles de acesso permitem que o projetista do programa oculte parte do tipo de dado abstrato dos clientes. Esses mesmos controles normalmente estão presentes nas classes das linguagens orientadas a objeto. As classes derivadas são outro tipo de cliente para que o acesso possa ser concedido ou recusado. Para levar isso em conta, algumas linguagens orientadas a objeto incluem uma terceira categoria de controle de acesso, freqüentemente chamada de protegida (*protected*), usada para fornecer acesso a classes derivadas e recusá-lo de outras classes. (Sebesta, 2000)

Além de herdar entidades de sua classe-pai, uma classe derivada pode acrescentar novas entidades e modificar métodos herdados. Um método modificado tem o mesmo nome, e freqüentemente o mesmo protocolo, de quando é uma modificação. Diz-se que o novo método sobrepõe-se (*override*) à versão herdada, que é então chamado de método sobreposto (*overriden*). O propósito mais comum de um método de sobreposição é fornecer uma operação específica a objetos de classe derivada, mas que não seja apropriada para objetos da classe-pai. Por exemplo, considere uma hierarquia de classes em que a classe-raiz descreve as características arquitetônicas gerais de catedrais góticas francesas. Esta classe-raiz, *French_Gohic*, tem um método para desenhar a fachada de uma catedral gótica francesa genérica. Em seguida, suponhamos que a classe *French_Gohic* tenha três derivadas, *Reims*, *Amien* e *Chartres*, cada uma das quais inclui um método para desenhar sua fachada particular. Tais versões de *draw* devem sobrepor-se ao método *draw* herdado da classe-pai. (Sebesta, 2000)

Classes podem ter dois tipos de métodos e dois tipos de variáveis. Os métodos e as variáveis mais comumente usados são chamados de **instância**. Todo objeto de uma classe tem seu próprio conjunto de variáveis de instância, as quais armazenam o estado do objeto. A única diferença entre dois objetos da mesma

classe é o estado de suas variáveis de instância. Os métodos de instância operam somente nos objetos da classe. As **variáveis de classe** pertencem a esta, não ao seu objeto, de maneira que há somente uma cópia da classe. Os **métodos de classe** podem executar operações nela, e possivelmente também nos objetos da mesma. Deste ponto em diante, serão ignorados os métodos de classe e as suas variáveis. (Sebesta, 2000)

Se uma classe criada por meio de herança tiver uma única classe-pai, o processo irá chamar-se **herança simples**. Se a classe tiver mais de uma classe-pai, o processo irá chamar-se **herança múltipla**. Quando um grande número de classes está relacionado pela herança simples, as relações entre elas podem ser mostradas em uma árvore de derivação. As relações de classes em uma herança múltipla podem ser mostradas em um grafo de derivação. (Sebesta, 2000)

O projeto de programa para um sistema orientado a objeto inicia-se com a definição de uma hierarquia de classes que descreva as relações dos objetos que preencherão o programa-solução. Quanto melhor essa hierarquia de classes combinar-se com o espaço de problema, mais natural será a solução completa. (Sebesta, 2000)

Uma desvantagem da herança como meio de aumentar a possibilidade de reutilização é que ela cria uma dependência entre as classes em uma hierarquia de herança. Isso trabalha contra uma das vantagens dos tipos de dados abstratos: o fato deles serem independentes uns dos outros. Obviamente, nem todos os tipos de dados abstratos devem ser completamente independentes. Mas, em geral, a independência dos tipos de dados abstratos é uma de suas características mais fortes. Porém, pode ser difícil, se não impossível, aumentar a capacidade de reutilização de tipos de dados abstratos sem criar dependências entre alguns deles. (Sebesta, 2000)

7.15.2.5 Polimorfismo e Vinculação Dinâmica

A terceira característica das linguagens de programação orientadas a objeto é um tipo de polimorfismo proporcionado pela vinculação dinâmica de mensagens a definições de método. Isso é suportado permitindo-se que alguém defina variáveis polimórficas do tipo da classe-pai que também são capazes de referenciar objetos de qualquer uma das subclasses da mesma. A classe-pai pode definir um método sobreposto por suas subclasses. As operações definidas por tais métodos são similares, mas devem ser personalizadas para cada classe da hierarquia. Quando esse método é chamado por meio da variável polimórfica, essa chamada é vinculada dinamicamente ao método da própria classe. Um propósito dessa vinculação dinâmica é permitir que os sistemas sejam mais facilmente estendidos tanto durante o desenvolvimento como durante a manutenção. Os programas podem ser escritos para executar operações em objetos-classe genéricos. Estas operações são genéricas em termo de que podem aplicar-se a objetos de qualquer classe relacionados por meio de derivação da mesma classe básica. Como um exemplo de vinculação dinâmica, considere o exemplo das catedrais. Se um programa que usa `French_Gothic` tiver uma variável polimórfica, `cathedral`, do tipo da `French_Gothic`, essa variável poderia referenciar objetos de `French_Gothic` e também objetos de qualquer uma das classes derivadas. Agora, quando `cathedral` for usada para chamar `draw` (definido em `French_Gothic` e em todos os seus descendentes), essa chamada será vinculada dinamicamente à versão correta de `draw`, escolhida pelo tipo ao qual a variável polimórfica está então referenciando. (Sebesta, 2000)

A vinculação dinâmica por meio de variáveis polimórficas é um conceito poderoso. Suponhamos que nosso exemplo das catedrais seja escrito em linguagem C. Os três exemplos de catedrais góticas francesas poderiam ser armazenados em variáveis de um tipo **`struct`**. Poderia haver uma única função `draw` que usasse uma instrução **`switch`** para chamar a função de desenho correta, baseando-se na catedral específica. Porém, nesse tipo de implementação, cabe ao programador chamar a versão apropriada da função de desenho. A

manutenção é muito mais fácil com a implementação orientada a objeto. Por exemplo. Suponhamos adicionar uma nova catedral à coleção, digamos, `Paris`. Isso nos obrigaria a modificar a construção **`switch`** na função de desenho geral, juntamente com quaisquer funções similarmente construídas para catedrais. No caso orientado a objeto, entretanto, adicionar uma nova catedral não tem nenhum efeito no código existente. (Sebesta, 2000)

Em muitos casos, o projeto de uma hierarquia de heranças resulta em uma ou em mais classes que estão tão altas na hierarquia que uma instanciação delas não faz sentido. Por exemplo, suponhamos que houvesse uma classe `building` como classe-pai ou como ancestral da `French_Gothic`. Provavelmente, não faria sentido ter um método `draw` implementado em `building`. Mas, uma vez que todas as suas classes descendentes devem ter esse método implementado, o protocolo (mas não o corpo) desse método é incluído em `building`. Esse método abstrato, muitas vezes, é chamado de **método virtual**. Além disso, qualquer classe que inclua pelo menos um método virtual é chamada de **classe virtual**. Essa última não pode ser instanciada, porque nem todos os seus métodos têm corpos. Qualquer subclasse de uma classe virtual que deva ser instanciada deverá oferecer implementações de todos os métodos virtuais herdados. (Sebesta, 2000)

7.15.3 Computando com uma Linguagem Orientada a Objeto

Toda computação em uma linguagem orientada a objeto pura é feita pela mesma técnica uniforme: enviando uma mensagem a um objeto para invocar um de seus métodos. Uma resposta a uma mensagem é um objeto que retorna o valor da computação do método. (Sebesta, 2000)

Um programa em execução em uma linguagem orientada a objeto pode ser descrito como uma simulação de uma coleção de computadores (objetos) que se comunicam entre si pelas mensagens. Cada objeto é uma abstração de um computador pelo fato dele armazenar dados e oferecer capacidades de

processamento para manipulá-los. Além disso, objetos podem enviar e receber mensagens. Em essência, essas são as capacidades fundamentais dos computadores: armazenar e manipular dados e comunicar-se. (Sebesta, 2000)

A essência da programação orientada a objeto é resolver problemas identificando os objetos do mundo real do problema e o seu processamento necessário, criando, então, simulações dos mesmos, seus processos e as comunicações necessárias entre eles. (Sebesta, 2000)

8 RESULTADOS

Os resultados estão organizados da seguinte forma. Primeiro são apresentadas estrutura e competências da AGERGS e do Núcleo de Informática. Na seqüência, os obtidos na análise da linguagem de programação Visual BASIC com destaque aos recursos e implementações relevantes na identificação dos paradigmas utilizados nas versões dessa linguagem, ao longo de sua evolução. Finalmente são apresentados os resultados da análise do SOA, abordando sua estrutura interna, as ferramentas, os recursos e os paradigmas de programação utilizados, com base nos arquivos de código fonte do software e a ferramenta de desenvolvimento Microsoft Visual BASIC versão seis. Inclui-se ainda algumas considerações a respeito de possíveis melhorias que podem ser implementadas nesse software, visando sua atualização para usufruir dos recursos da programação orientada a objeto disponíveis na versão do VB .NET, fato que provavelmente poderá aumentar a produtividade e reduzir as rotinas administrativas do NIN.

8.1 AGERGS

A Agência Estadual de Regulação dos Serviços Públicos Delegados do Rio Grande do Sul é uma autarquia criada em 09 de janeiro de 1997 na forma da Lei n°10.931, dotada de autonomia financeira, funcional e administrativa, com sede na Capital do Estado. O objetivo da agência é garantir a qualidade dos serviços públicos (energia, telecomunicações, transporte etc) oferecidos aos usuários pelos concessionários do setor privado e o equilíbrio econômico e financeiro dos contratos entre o poder concedente (Governo) e as empresas concessionárias.

Compete à Agergs a regulação dos serviços públicos delegados prestados no estado do rio grande do sul:

a) garantir a aplicação do princípio da isonomia no acesso e uso dos serviços públicos;

b) homologar os contratos e demais instrumentos celebrados, assim como seus aditamentos ou extinções;
c) zelar pelo fiel cumprimento dos contratos;
d) propor o aditamento ou a extinção dos contratos em vigor;
e) fixar, reajustar, revisar, homologar ou encaminhar ao ente delegante, tarifas, seus valores e estruturas;
f) buscar a modicidade das tarifas e o justo retorno dos investimentos;
g) requisitar à Administração, aos entes delegantes ou aos prestadores de serviços públicos delegados as informações convenientes e necessárias ao exercício de sua função regulatória;
h) permitir o amplo acesso às informações sobre a prestação dos serviços públicos;
i) orientar a confecção dos editais de licitação e homologá-los, objetivando à delegação de serviços públicos no Estado;
j) propor novas delegações de serviços públicos no Estado;
k) moderar, dirimir ou arbitrar conflitos de interesse;
l) cumprir e fazer cumprir a legislação específica relacionada aos serviços públicos;
m) fiscalizar a qualidade dos serviços;
n) aplicar sanções decorrentes da inobservância da legislação vigente ou por descumprimento dos contratos;
o) fiscalizar a execução do Programa Estadual de Concessão Rodoviária (PECR).

As áreas de atuação estabelecidas em lei são as de saneamento, energia elétrica, rodovias, telecomunicações, portos e hidrovias, irrigação, transportes intermunicipais de passageiros, estações rodoviárias, aeroportos, distribuição de gás canalizado e inspeção de segurança veicular.

A agência possui uma estrutura constituída de órgãos de Direção Superior, de Assistência e Assessoramento Direto e Imediato, de Execução e de Apoio Administrativo, contando atualmente com recursos humanos de sessenta e uma

pessoas no total, sendo que três exercem suas funções no NIN. Essa estrutura esta organizada em:

a) Órgãos de Direção Superior: Conselho Superior composto por sete Conselheiros; e Presidência exercida por um dos Conselheiros;
b) Órgãos de Assistência e Assessoramento Direto e Imediato: Gabinete da Presidência; Assessoria de Comunicação Social; e Secretaria Executiva;
c) Órgão de Execução: Diretoria-Geral; Diretoria de Qualidade dos Serviços; Diretoria de Tarifas e Estudos Econômico-Financeiros; e Diretoria de Assuntos Jurídicos;
d) Ouvidoria;
e) Órgão de Apoio Administrativo: Gabinete Administrativo; Núcleo Setorial de Administração; Núcleo Setorial de Finanças; Núcleo Setorial de Informática; Núcleo Setorial de Recursos Humanos.

O artigo 18º do Regimento Interno da agência estabelece que compete ao Núcleo de Informática:

a) coordenar a distribuição dos equipamentos de informática;
b) orientar a utilização dos equipamentos;
c) controlar os contratos de locação de “hardware” e “software”;
d) realizar manutenção preventiva dos equipamentos;
e) providenciar na atualização dos aplicativos e na conservação dos equipamentos;
f) fiscalizar a utilização dos equipamentos;
g) coordenar a implantação de rede interna e a conexão em linha dedicada na “Internet”;
h) coordenar a implantação do Banco de Dados da AGERGS;
i) coordenar a informatização do sistema de atendimento ao usuário através do Sistema de Ouvidoria;
j) exercer outras atividades correlatas ou que lhe venham a ser atribuídas.

8.2 LINGUAGEM DE PROGRAMAÇÃO VISUAL BASIC

8.2.1 Linguagem de Programação BASIC

A linguagem BASIC (*Beginners All-purpose Symbolic Instruction Code*) se refere a uma família de linguagens de programação de alto nível. Ela foi originalmente projetada em 1963 com propósitos didáticos para fornecer aos estudantes acesso aos computadores. Nesta época, quase todos os computadores faziam uso de software customizado, o que era alguma coisa que somente cientistas e matemáticos estavam dispostos a fazer. (Wikipedia, 2007)

A linguagem BASIC passou a ter seu uso bastante difundido nos microcomputadores das décadas de 1970 e 1980. No entanto, a partir da segunda metade da década de 1980, novos computadores de maiores capacidade e recursos foram lançados e muitos programas foram desenvolvidos, tornando a programação menos importante para a maioria dos usuários. Desta forma, a linguagem BASIC vai perdendo importância até ressurgir com a introdução do Visual BASIC. É difícil considerar esta linguagem como sendo BASIC, a única similaridade significativa aos antigos dialetos BASIC é a sintaxe familiar. (Wikipedia, 2007)

O nome BASIC também se refere a uma grande quantidade de linguagens de programação derivadas do original. Provavelmente, existem mais variações dessa linguagem do que qualquer outra. Comparar o BASIC original com as versões atuais, principalmente com Visual BASIC e Visual BASIC .NET, não faz muito sentido devido às grandes alterações ocorridas durante sua evolução. (Wikipedia, 2007)

A linguagem de programação BASIC pode ser classificada como uma linguagem imperativa de alto nível, pertencente à terceira geração. Pelo fato de ter sido fortemente baseada no FORTRAN, ela é normalmente interpretada e, originalmente, não estruturada. Com o tempo ela evoluiu e foram criados recursos para dar suporte à programação estruturada. (Wikipedia, 2007)

8.2.2 Linguagem de Programação Visual BASIC

A análise da linguagem de programação Visual BASIC se divide principalmente em dois segmentos. O primeiro se refere principalmente ao VB6 – Visual BASIC versão seis – lançado no ano de 1998 e o segundo ao VB .NET – Visual BASIC ponto NET – lançado em 2003. Embora esta última apresente ainda alguma semelhança com a anterior, essas duas versões não são mais compatíveis. O VB6 é ainda bastante utilizado, sendo o momento atual de transição, devido ao fato de que a empresa proprietária da linguagem, em breve, não mais oferecerá suporte a mesma, segundo a própria Microsoft (2007), a partir de março de 2008. Já o VB .NET é uma nova plataforma de desenvolvimento de software com suporte a programação orientada a objeto que tem a intenção de ser a atualização natural do VB6. Ainda não existe consenso sobre o assunto e somente o tempo vai nos trazer a resposta.

8.2.2.1 Visual BASIC: do VB1 ao VB6

A linguagem de programação Visual BASIC (VB) foi desenvolvida pela empresa Microsoft, a primeira palavra “Visual” refere-se ao método usado para criar a interface gráfica de usuário (GUI – *Graphical User Interface*) e a segunda à linguagem BASIC, da qual foi derivada. Ela possui um ambiente de desenvolvimento integrado (IDE – *Integrated Development Environment*) para desenvolver aplicativos baseados no sistema operacional *Windows*, fazendo uso dos modelos de programação COM (*Component Object Model*) da Microsoft. O ambiente de trabalho IDE do VB integra em um único ambiente várias funções diferentes, tais como criação, edição, compilação e depuração. (Microsoft, 1998; Wikipedia, 2007)

A figura 3 mostra o ambiente IDE do Visual BASIC versão 6.

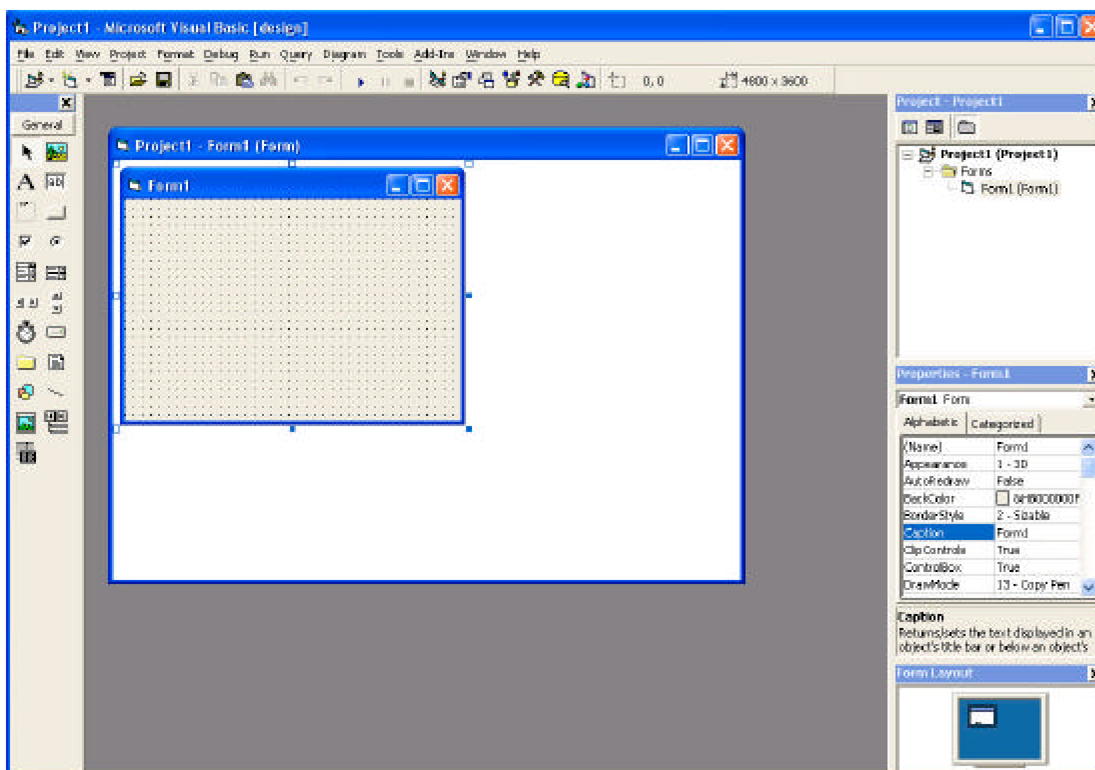

Figura 3: IDE do Visual BASIC 6
Fonte: Microsoft Visual Basic 6 (1998)

A linguagem VB viabiliza o uso do modelo de desenvolvimento rápido de aplicação (RAD – *Rapid Application Development*) e do tipo de interfaces GUI. Ela é uma linguagem coletora de lixo (*garbage collected*) que usa contagem referenciada. Também possui uma extensa biblioteca de objetos e um suporte básico de orientação a objeto. Em suas primeiras versões, não permitia acesso a bancos de dados, sendo voltada apenas para iniciantes. No entanto, devido a sua aceitação entre as empresas, logo foram incorporadas tecnologias como DAO (*Data Access Objects*), RDO (*Remote Data Objects*), e ADO (*ActiveX Data Objects*) permitindo o acesso a bancos de dados. (Microsoft, 1998; Wikipedia, 2007)

A programação em VB é essencialmente uma combinação de três atividades. Uma é a organização visual dos componentes (formulários e/ou controles), outra a especificação dos atributos e ações desses componentes, e uma terceira que é a de escrever linhas de código para adicionar maior funcionalidade ao software. Projetada para ser de fácil uso e aprendizado. Permite o desenvolvimento de

aplicativos simples com as interfaces GUI e também apresenta uma flexibilidade para desenvolver aplicações mais complexas. (Microsoft, 1998; Wikipedia, 2007)

Alguns problemas de performance foram experimentados nas primeiras versões, mas o lançamento de computadores mais rápidos e da implementação do recurso de compilação de código nativo, tornou esse fato insignificante. Apesar dos programas serem compilados em código nativo executáveis a partir da versão cinco, eles exigem a presença de algumas bibliotecas durante o tempo de execução. Essas bibliotecas são incluídas por padrão no Windows 2000 e superior, mas para versões mais antigas do Windows elas devem ser distribuídas junto com o arquivo executável. (Microsoft, 1998; Wikipedia, 2007)

No ambiente de programação do VB, os componentes são criados com as técnicas de pegar e arrastar (*drag and drop techiniques*). Uma caixa de ferramentas agrupa esses componentes e é utilizada, por exemplo, para criar controles (botões, caixas de texto, etc.) em um formulário (ou janela). Existem atributos e controladores de eventos associados aos controles. Os valores padrões são definidos quando o controle é criado, mas podem ser alterados durante os tempos de criação. Muitos valores de atributos podem ser modificados durante o tempo de execução, com base nas ações dos usuários ou mudanças no ambiente. Por exemplo, linhas de código podem ser inseridas no controlador de evento **mudar tamanho** de um formulário para reposicionar um controle, de tal forma que este fique centralizado no formulário; ou aquelas podem ser inseridas no procedimento do controlador de evento **apertar tecla** (keypress) de um controle caixa de texto, para que o programa mude automaticamente o tamanho do texto que esta sendo digitado pelo usuário, ou previna que certos caracteres sejam inseridos. (Microsoft, 1998; Wikipedia, 2007)

A programação em VB permite desenvolver projetos de arquivos executáveis padrões (arquivos .exe), de controles ActiveX e de arquivos de biblioteca de ligação dinâmica (DLL – *Dynamic-link library*), bem como fazer interface de sistemas de base de dados em rede. Esses projetos são desenvolvidos e executados na plataforma do sistema operacional Windows. A DLL é o conceito de bibliotecas compartilhadas adotado pela Microsoft na implementação de seu sistema operacional. Anteriormente chamados de controles de vinculação e incorporação de objetos (OLE – *Object Linking and Embeldding*), os controles ActiveX permitem

utilizar a funcionalidade oferecida por outros aplicativos, facilitando a integração entre vários programas. (Microsoft, 1998; Wikipedia, 2007)

A linguagam VB pode ser classificada como imperativa de alto nível, a qual permite a abstração de processos. Ela é uma linguagem dirigida por eventos (*event driven*). Segundo Wikipedia (2007), a programação dirigida por eventos é um paradigma de programação no qual o fluxo de programa é determinado pela ação do usuário (como, clicar o mouse e pressionar tecla) ou pelas mensagens de outros programas. No entanto, a versão VB6 e anteriores não trazem suporte a programação orientada a objetos, nem fazem implementação de herança.

8.2.2.2 Visual BASIC .NET

A linguagem de programação Visual BASIC .NET (VB .NET) é considerada uma evolução do VB. Significativas mudanças foram implementadas tornando essa versão incompatível com as anteriores. Fato que gerou muita controvérsia e causou uma divisão na comunidade de desenvolvedores. (Wikipedia, 2007)

A linguagem VB .NET é uma linguagem de programação imperativa que oferece suporte à programação orientada a objeto. É implementada com base no Microsoft *.NET Framework*, o qual fornece uma biblioteca de classes que inclui interfaces de usuários, acesso a dados, conectividade com banco de dados, criptografia, algoritmos numéricos, comunicação de redes, entre outras. Os programas escritos para o *.NET Framework* são executados em um ambiente de software que administra as necessidades dos programas durante o tempo de execução. Este ambiente de execução, que também é parte do *.NET Framework*, é denominado CLR (*Common Language Runtime*). (Wikipedia, 2007)

A plataforma .NET utiliza o ambinte de programação Visual Studio, o qual é um grupo de ferramentas de desenvolvimento para construir aplicações ASP.NET da internet, Serviços XML da internet e aplicativos de console (*desktop*). As linguagens do Visual Studio (Visual Basic, Visual C++, Visual C#, e Visual J#) usam

todas o mesmo ambiente integrado de desenvolvimento (IDE), o qual permite que elas dividam ferramentas e recursos na criação de soluções de linguagem mistas. Adicionalmente, essas linguagens elevam a funcionalidade do .NET Framework, o qual oferece acesso a tecnologias chaves que simplificam o desenvolvimento de aplicações ASP da internet e serviços XML da internet. (Microsoft, 2005)

O *.NET Framework* é um compontente integrante do Windows que fornece suporte para construir e executar a nova geração de aplicativos e serviços da Web XML.

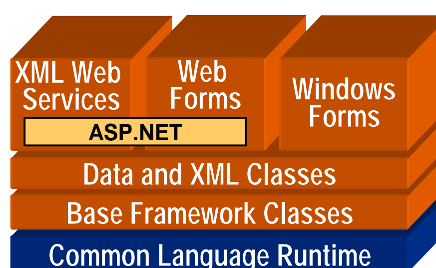

Figura 4: Plataforma .NET
Fonte: Microsoft, 2001

O *.NET Framework* é projetado para preencher os seguintes objetivos:

a) Fornecer um consistente ambiente de programação orientado a objeto se o código do objeto é armazenado e executado localmente, executado localmente mas distribuído na Internet, ou executado remotamente;

b) Fornecer um ambiente de código-execução que minimiza os conflitos de preparação e versão de software;

c) Fornecer um ambiente de código-execução que promove execução de código seguro;

d) Fornecer um ambiente de código-execução que elimina os problemas de performance de ambientes interpretados ou desenvolvidos com linguagens *script*;

e) Fazer o desenvolvedor experimentar consistencia entre uma variedade de tipos de aplicações,, tal como aplicativos baseados no Windows e aplicativos baseados na Internet;

f) Construir toda comunicação nos padrões da industria para garantir que o código baseado no *.NET Framework* possa integrar com qualquer outro código. (Microsoft, 2005)

O *.NET Framework* tem dois principais componentes: a linguagem comum de tempo de execução (CLR - *common language runtime*) e a biblioteca de classes .NET Framework. O CLR é a base do *.NET Framework*. Considerado como um agente que administra código no tempo de execução, fornecendo um núcleo de serviços tais como administração de memória, administração encadeada, e remota, enquanto também exigindo segurança de tipo restrito e outras formas de acuracidade de código que oferece segurança e robustez. De fato, o conceito de administração de código é um princípio fundamental do tempo de execução. Código que é direcionado para o tempo de execução é conhecido como código administrado, enquanto código que não é direcionado ao tempo de execução é denominado de código não administrado. (Microsoft, 2005)

A biblioteca de classes, o outro principal componente do *.NET Framework*, é uma abrangente, coleção orientada a objeto de tipos reusáveis que podem ser usados para desenvolver aplicativos extendendo-se de aplicações tradicional linha de comando ou aplicações GUI até aplicações baseadas nas últimas inovações fornecidas pelo ASP.NET, tal como Formulários da Internet e serviços XML da internet. O *.NET Framework* permite criar aplitivos desenvolvidos para execução do lado do cliente (*client-side applications*) e aplicativos para o lado do servidor (*server-side applications*). Com o *.NET Framework*, por exemplo, é possível desenvolver os seguintes tipos de aplicativos e serviços: aplicativos de console, de Windows GUI (Windows *Forms*), de ASP.NET e de serviços tanto de Windows quanto de XML de internet. (Microsoft, 2005)

8.2.2.3 Evolução do Visual BASIC

A evolução do Visual BASIC é descrita brevemente, como segue:

a) a empresa Microsoft criou um projeto denominado *Thunder* com o propósito de desenvolver o Visual BASIC;
b) foi lançado VB 1.0 para Windows, em 1991;
c) em setembro de 1992, foi lançado VB 1.0 para DOS, versão não muito compatível com o VB para Windows e que deu origem a próxima versão do compilador DOS-based BASIC da Microsoft, o QuickBASIC e o BASIC *Professional Development System*. A interface era textual, usando caracteres ASCII extendidos;
d) VB 2.0 foi laçado em novembro de 1992, o ambiente de programação foi mais fácil de usar, e sua rapidez foi melhorada. Os formulários tornaram-se objetos instanciáveis, colocando, desta forma, os conceitos fundamentais de módulos de classe, os quais foram mais tarde incorporados no VB4;
e) VB 3.0 foi lançado em 1993, nas versões *Standard* e *Professional* e incluiu a versão 1.1 do Microsoft *Jet Database Engine* que podia ler e gravar banco de dados *Jet* (ou do aplicativo *Access*);
f) em 1995, foi lançado o VB 4.0, a primeira versão que criava programas Windows 32-bit tanto quanto 16-bit. Ela também introduziu a facilidade de escrever classes não GUI no VB;
g) o VB 5.0 foi lançado em 1997, exclusivamente para versões do Windows de 32-bit. Programadores que preferiam escrever programas 16-bit foram capazes de importar programas escritos no VB 4.0 para o VB 5.0, e os programas do VB 5.0 podiam facilmente ser convertidos com VB 4.0. VB 5.0 também introduziu a facilidade de criar controles customizados, tão bem quanto à facilidade de compilar para códigos executáveis nativos de Windows, acelerando execução de códigos de cálculos intensivos;
h) o VB 6.0 lançado em 1998 melhorou em um número de áreas, incluindo a facilidade de criar aplicativos para internet. VB6 está atualmente

programado para entrar para a “fase de não suporte” (“non-supported phase”) da Microsoft em 2008. Em resposta, a comunidade de usuários expressou sua preocupação, procurando manter o produto ativo. A Microsoft por enquanto tem se recusado mudar de posição nesta questão;

i) Visual BASIC .NET (VB 7) foi lançado junto com o Visual C# e o ASP.NET em 2002. A linguagem de programação C# foi bastante divulgada como uma resposta da Microsoft ao Java, enquanto o VB .NET não recebeu muita atenção. Esta versão do VB trouxe uma linguagem poderosa mas muito diferente, com desvantagens em algumas áreas, incluindo o tempo de execução para empacotar uma solução e um aumento do uso de memória;

j) Visual BASIC .NET 2003 (VB 7.1) foi lançado com uma versão 1.1 do .NET Framework. Novos recursos foram incluídos e deram suporte ao .NET Compact Framework e um melhor suporte para atualização do VB. Também melhorou a performance e a confiabilidade da .NET IDE (particularmente o compilador interno) e o tempo de execução;

k) Visual BASIC 2005 (VB 8.0) é a nova versão do Visual BASIC .NET, a porção .NET foi retirada do título. Para essa versão, foram adicionados novos recursos, os quais têm a intenção de reforçar o foco do Visual Basic .NET como uma plataforma de desenvolvimento rápido de aplicativos, o que o diferencia mais ainda do C#. (Wikipedia, 2007)

8.2.2.4 Linguagem Derivadas do Visual BASIC

A empresa Microsoft desenvolveu algumas linguagens de programação *Scripting* derivadas do Visual BASIC:

a) *Visual Basic for Applications* (VBA): incluída em muitos dos aplicativos da Microsoft (Microsoft Office), e também em muitos produtos de terceiros tais como AutoCAD e WordPerfect Office 2002;
b) *VBScript*: uma linguagem padrão para ASP (*Active Server Pages*) e pode ser usado em *Windows scripting* e páginas da internet do lado do cliente. (Wikipedia, 2007)

As linguagens *Scripting* são interpretadas comando por comando a cada vez que são executadas, diferentemente dos programas que são compilados permanentemente em arquivos binários executáveis. Existe pequena inconsistência na maneira como o VBA é implementado nas diferentes aplicações, mas é amplamente a mesma linguagem da versão seis do VB. Embora relembre o VB na sintaxe, *VBScripting* não é uma linguagem separada e é executada pelo uso de bibliotecas *Windows Script Host* tal como o VB faz uso de *VB runtime*. Compilado e executado no *.NET Framework*, o VB .NET não é compatível com o VB6. Está disponível uma ferramenta de conversão automatizada, mas para alguns projetos essa conversão se torna impossível. ASP e *VBScript* não devem ser confundidas com ASP.NET, o qual faz parta da plataforma .NET e faz uso de *.NET Common Language Runtime*. (Wikipedia, 2007)

8.3 SISTEMA DE OUVIDORIA DA AGERGS

Como descrito anteriormente, um sistema baseado em computador é um conjunto de elementos organizados para executar uma determinada tarefa. Com o propósito de transformar informações, esses elementos freqüentemente incluem software, hardware, pessoas, banco de dados, documentação e procedimentos.

O sistema baseado em computador de interesse do presente estudo é o da Ouvidoria da Agergs, apresenta as seguintes características.

A ouvidoria da AGERGS está à disposição dos usuários, prestadores de serviços públicos concedidos e Governo, com o propósito de dirimir dúvidas e

intermediar soluções nas divergências entre as partes. Atualmente, há um banco de dados que reúne todas as reclamações encaminhadas pelos usuários, sua tramitação junto às empresas operadoras dos serviços, bem como as soluções encontradas. No presente momento, a Ouvidoria conta com uma equipe de sete pessoas efetivas e quatro em regime de estágio. Os recursos de hardware são doze microcomputadores, conectados à rede da agência, sendo que um fica a disposição dos usuários dos serviços públicos. O banco de dados do setor está localizado fisicamente em um servidor de arquivos da rede.

Um dos programas de computador utilizados pela Ouvidoria é o Sistema de Ouvidoria da Agergs (SOA) que tem a finalidade de oferecer suporte aos procedimentos de atendimento às reclamações. Iniciado em 2000, o projeto do SOA ao longo desses anos vem evoluindo e sendo modificado para melhor suprir as necessidades do setor, implementado uma interface de cadastro, atualização e manutenção do banco de dados de reclamações. Esse software foi desenvolvido pelo Núcleo de Informática da Agergs (NIN) com base nos recursos e linguagem de programação do Microsoft Visual BASIC 6.0 *Professional Edition*. As informações do SOA são organizadas e arquivadas em um banco de dados do aplicativo Access fornecido no Microsoft Office 98.

As informações referentes aos atendimentos realizados que correspondem as áreas de atuação da Agergs, são recebidas e cadastradas no SOA, exceto os atendimentos de Energia Elétrica que possui um tratamento diferenciado, seguindo os procedimentos e regramentos adotados pela ANEEL. O acesso ao SOA com permissão de cadastro e atualização é dado à oito pessoas da Ouvidoria responsáveis pelas informações e à duas do NIN para suporte.

O banco de dados possui oito tabelas com as informações de reclamações, cadastro das empresas, áreas de atuação, tipo de assunto, modo de recebimento, relação de municípios e dos usuários. A tabela de reclamações conta com 2787 registros de reclamações, pedidos de informação e atendimentos cadastrados.

Existem atualmente três versões do SOA: a versão do SOA propriamente dita, que é a versão completa utilizada na ouvidoria; a de consulta, utilizada por outros setores da agência para consulta das Solicitações de Ouvidoria e uma versão para distribuição externa, para outras instituições que solicitam uma cópia do

software. O SOA está instalado em oito computadores com a versão completa e distribuídos pelos setores da Agergs em 23 máquinas com a versão que permite somente consulta ao banco de dados. Todas as instalações e configurações do software são realizadas pelos técnicos do NIN.

A partir deste ponto, será apresentada a estrutura do SOA implementada com os recursos do VB6.

O SOA faz uso das seguintes bibliotecas, denominadas de Referências, que são as Bibliotecas DLLs e objetos disponíveis no VB6 ou de outros aplicativos:

a) *Visual Basic for Applications;*
b) *Visual Basic runtime objetcts and procedures;*
c) *Visual Basic objects and procedures;*
d) *Microsoft OLE Automation;*
e) *Microsoft DAO Object Library;*
f) *Microsoft Data Binding Collection;*
g) *Microsoft Data Environment Instance;*
h) *Microsoft ActiveX Data Objects Library;*
i) *Microsoft Data Report Designer;*
j) *Microsoft Data Formatting Object Library.*

Utiliza também alguns Componentes, tais como denominado no VB6 de controles e designers (ou criadores). Os controles implementados no SOA são basicamente os necessários para fazer conexão com o banco de dados e mostrar as informações ao usuário, a saber:

a) *Microsoft ADO Data Control (OLEDB);*
b) *Microsoft Common Dialog Control;*
c) *Microsoft Data Bound List Controls;*
d) *Microsoft DataGrid Control (OLEDB);*
e) *Microsoft DataList Controls (OLEDB);*
f) *Microsoft FlexGrid Control;*
g) *Microsoft Masked Edit Control.*

Os designers implementados por meio das DLLs disponíveis são: *Data Environment* (criador de conexão com banco de dados) e *Data Report* (criador de relatórios).

A interface é a parte visual de uma aplicação com a qual o usuário interage. Os Formulários (*Form* ou janela) e controles são objetos usados para criar a interface. Os objetos expõem propriedades que definem sua aparência, métodos que definem seu comportamento, e eventos que definem as suas interações com o usuário. Os objetos são customizados ajustando as propriedades e escrevendo linhas de código para responder aos eventos. A customização do objeto é feita de forma a atender os requisitos do programa. Os controles são os objetos que estão contidos dentro de objetos *Form*. Cada tipo de controle possui suas próprias propriedades, métodos e eventos, sendo cada tipo adequado a uma determinada finalidade. (Microsoft, 1998)

A interface do SOA com o usuário é feita sob o estilo de múltiplos documentos (MDI – *multiple-document interface*), possibilitando mostrar vários formulários ao mesmo tempo dentro do formulário principal. O formulário principal conta com uma estrutura de menus e submenus para acesso aos demais formulários do programa.

Quando o SOA é inicializado, o primeiro formulário mostrado ao usuário é o Login, verificando as informações de usuário e senha por meio de uma consulta ao banco de dados, usando o objeto DAO. Passada esta etapa é inicializado o formulário principal, exibindo os menus de acesso aos recursos do programa.

No total são sete formulários denominados: principal, cadastro SO, consulta SO, consulta usuário, login, relatórios e sobre. Possui um módulo para definição de variáveis globais. Com os Designers foram criados sete instâncias: um *DataEnvironment* para conexão com o banco de dados e seis *Data Report* para implementação dos relatórios. O *DataEnvironment* faz a conexão com o banco de dados do arquivo em Access98 usando o provedor *Microsoft Jet OLE DB Provider*. Nesse ambiente, são feitas consultas SQL à tabela de reclamações, fornecendo informações aos relatórios. Cada relatório faz uso do *DataEnvironment* e do *DataReport* para consulta ao banco de dados e para mostrar os relatórios respectivamente. Os relatórios podem ser para visualização na tela ou para impressão.

O *ADO Data control* (é um controle gráfico que pode ser criado no formulário) usa o *Microsoft ActiveX Data Objects ADO* para criar conexões entre controles *data-*

bound e *data providers*. Um controle *Data-bound* é qualquer controle que possui uma propriedade *DataSource* que pode ser conectado a uma fonte de dados *DataSource*. Um controle *Data providers* pode ser qualquer fonte escrita para as especificações de OLE DB. Um formulário (objeto *Form*) é uma janela que faz parte da interface de usuário de um aplicativo, onde são criados os outros controles da interface de usuário. O formulário MDI é um tipo especial de formulário, contendo os outros formulários chamados formulários filhos MDIChild. (Microsoft, 1998)

Os formulários (*forms*) do SOA fazem uso não apenas do controle *ADO Data Control* para fazer a conexão ao banco de dados, o qual faz uso dos objetos ADO (*ActiveX Data Objects*), mas também dos seguintes controles:

a) Caixas de texto (*TextBox control*) permite mostrar informações e entrada de dados inseridos pelo usuário;
b) *DataCombo* controle é um *data-bound combo box* que é automaticamente populado do campo em que está ligado a fonte de dados, e opcionalmente atualiza o campo em uma tabela relacionada em outra fonte de dados *data source*;
c) Controle *CheckBox* que oferece ao usuário uma opção de Verdadeiro/False ou Sim/Não;
d) Controle Botão de ação (*CommandButton*) é um controle para iniciar, interromper ou finalizar um processo;
e) Controle *Label* é um controle gráfico para exibir texto que o usuário não pode alterar diretamente no formulário;
f) *Masked Edit Control* fornece restrições de entrada de dados tanto quanto dados formatados de saída (como formatos de data, hora);
g) *OptionButton* exibe uma opção que pode ser ligada ou desligada, normalmente usados em um grupo de opções para exibir opções a partir das quais o usuário seleciona apenas uma;
h) *DataGrid* que mostra e permite a manipulação de informações de uma série de linhas e colunas representando os registros e campos de um objeto registro (*Recordset*).

Algumas considerações a respeito do SOA são feitas a seguir.

O NIN fez uso dos seguintes paradigmas de desenvolvimento de software. A estrutura interna dos procedimentos do SOA foram desenvolvidos com base no paradigma de **programação estruturada**. As implementações de conexão de banco de dados, de criação de relatórios e de interface com os usuários foi empregado o paradigma de programação dirigida por eventos, que é o paradigma implementado pela linguagem Visual BASIC do VB6.

A seguir, é aprestada a situação atual do SOA e das rotinas administrativas que envolvem a manutenção desse software, bem como algumas considerações sobre as possíveis implementações de melhorias do mesmo.

Com relação ao SOA, o NIN precisa fazer o gerenciamento das instalações e configurações do software. O banco de dados do SOA está localizado no servidor de arquivos, uma cópia de segurança desse servidor é feita diariamente. Em todas as conexões (objetos ADO e *Data Environment*) implementadas no software precisam fazer referência a unidade do servidor e ao nome do arquivo do banco de dados. Uma alteração de nome ou local desse arquivo exige que sejam corrigidas as propriedades das conexões, com intuito de refletir essa alteração.

As correções, novas implementações e alterações nas configurações do SOA requer à reinstalação do software em todas as máquinas com a nova versão, tanto da completa como de consulta.

Além disso, o SOA permite fazer instâncias dos objetos *form* possibilitando ao usuário inicializar várias vezes o mesmo objeto *form* (por exemplo, abrir e trabalhar com várias janelas do formulário consulta); utiliza objetos intrínsecos do VB6 e de objetos *ActiveX* das bibliotecas disponíveis; e, não interage com outros programas, nem permite usar os recursos de copiar e colar as informações de seus formulários. Os principais formulários do SOA são mostrados nos anexos.

Algumas melhorias que podem ser observadas para uma revisão e atualização deste projeto:

a) Adotar uma única tecnologia de acesso ao banco de dados;
b) Criar códigos para os registros nas tabelas;
c) Atualizar para a versão mais recente do VB;
d) Utilizar os recursos de internet para interface com os usuários;
e) Utilizar os recursos de ASP Active Server Pages;

f) Fazer mais uso de consultas SQL para acesso ao banco de dados, utilizando parâmetros;
g) Atualizar o banco de dados para uma versão mais recente do Access ou outro servidor de banco de dados;
h) Unificar as versões de consulta e cadastro, permitindo acesso aos usuários as suas respectivas tarefas de consulta e cadastro por meio de mecanismos de restrição de acesso.

Como consideração final, constatou-se que o SOA é um software que atende as necessidades da Ouvidoria, é confiável e seguro. No entanto, percebe-se que com os recursos atualmente disponíveis de ferramentas de desenvolvimento voltadas para internet, o pode ser atualizado para melhor atender as novas necessidades da Ouvidoria em particular e da Agergs como um todo.

9 CONCLUSÃO

Este trabalho realizou um estudo de caso da AGERGS, agência de regulação estadual, com o propósito de identificar o processo de desenvolvimento de software do Núcleo de Informática da Agergs. Para tanto, foi abordado o paradigma de desenvolvimento de software da programação orientada a objeto. De forma sintética, foram abordados os temas: a evolução e arquitetura dos computadores; a evolução e as metodologias de programação de software; a evolução e as características das linguagens de programação; e as estruturas e metodologias de desenvolvimento de software que abrange a Engenharia de Software. Mais especificamente, foi feita uma análise do SOA – Sistema de Ouvidoria da Agergs, software desenvolvido pelo Núcleo de Informática dessa agência, procurando investigar tanto as características da linguagem de programação utilizada, quanto o paradigma de desenvolvimento empregado nesse sistema. Os resultados evidenciaram que o software SOA foi desenvolvido com base na linguagem de programação Visual Basic. A versão do Visual Basic adotada no desenvolvimento do SOA é dirigida por eventos e faz uso de objetos. No entanto, uma versão mais recente dessa linguagem é orientada a objeto e disponibiliza recursos para desenvolvimento de programas voltados para a internet. Foram apresentadas as características de implementação do SOA, bem como sugestões de algumas possibilidades de melhoria e atualização. Os resultados sugerem uma atualização desse software para a nova versão da linguagem Visual BASIC, beneficiando-se da programação orientada a objeto e dos recursos disponíveis da plataforma *.NET Framework*. Finalmente, ressalta-se que não é de todo improvável que o presente estudo possa servir, de um modo geral, como referência para o desenvolvimento de programas com orientação a objeto, e mais especificamente, como suporte para os projetos do Núcleo de Informática.

REFERÊNCIAS

COAD, Peter e YOURDON, Edward. **Projeto Baseado em Objetos**. Ed. Rio de Janeiro: Campus, 1993. 195 p.

COLEMAN, Derek et al. **Desenvolvimento orientado a objetos : o método fusion.** Rio de Janeiro : Campus, 1996. 389 p.

FERGUSON, Scott. **The Birth of Visual Basic**. Disponível em <http://www.forestmoon.com/BIRTHofVB/BIRTHofVB.html>. Acesso em julho 2007.

FIANI, R. **Teoria da Regulação Econômica: Estado Atual e Perspectivas Futuras.** Rio de Janeiro: IE/UFRJ, 1998. Texto para Discussão n. 423. Disponível em: <www.ie.ufrj.br/grc/pdfs/teoria_da_regulacao_economica.pdf>. Acesso em: abril 2007.

GOVERNO DO ESTADO DO RIO GRANDE DO SUL. **LEI Nº 10.931 – Cria a Agência Estadual de Regulação dos Serviços Públicos Delegados do Rio Grande do Sul - AGERGS e dá outras providências**, 1997. Disponível em: <http://www.agergs.rs.gov.br/bibliot/legis/leis.htm#agergs>. Acesso em julho 2007.

GOVERNO DO ESTADO DO RIO GRANDE DO SUL. **Decreto Estadual 39.061 - Regimento Interno, 1998**. Disponível em: <http://www.agergs.rs.gov.br/bibliot/legis/leis.htm#agergs>. Acesso em julho 2007.

KANN, Zevi. Histórico e futuro das agências de reguladoras. **Regulação, defesa da concorrência e concessões**. Rio de Janeiro : Editora FGV 2002, pg.45-50.

MEIRELLES, Fernando de Souza. **Informática – Novas Aplicações com Microcomputadores**. 2. Ed. São Paulo: Makron Books, 1994.

MICROSOFT. **Visual Basic: Guia do Programador**. Microsoft Corporation, USA. 1998. 1072 páginas.

MICROSOFT. MSDN **Microsoft Visual Studio 2005 Documentation**. Microsoft Corporation, USA. 2005

MICROSOFT. **.NET Framework - Technical Overview**. Microsoft Corporation, USA. 2001. Disponível em:
<http://msdn2.microsoft.com/pt-br/netframework/aa497336.aspx>. Acesso em julho 2007.

MICROSOFT. **Product Family Life-Cycle Guidelines for Visual Basic 6.0**. Microsoft Corporation, USA. Disponível em:
<http://msdn2.microsoft.com/en-us/vbrun/ms788707(d=printer).aspx>. Acesso em: julho 2007.

MICROSOFT. Visual Basic 6 and Visual Basic .NET: Differences. **Free Book - Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET**. Microsoft Corporation, USA, 2007. Disponível em:
<http://msdn.microsoft.com/vbrun/staythepath/additionalresources/upgradingvb6/chapter2.pdf>. Acesso em: julho 2007.

PECI, Alketa e CAVANCANTI, Bianor Scelza. Reflexões sobre a autonomia do órgão regulador: análise das agências reguladoras estaduais. **Revista de Administração Pública**. Rio de Janeiro 34(5):99-118, Set. /Out . 2000

PEREIRA, Luiz Carlos Bresser. A Reforma do Estado dos anos 90: Lógica e Mecanismos de Controle. **Cardernos do MARE**, Caderno 1, Brasília –DF 1997.

PIRES, J. C. L; PICCININI, M. S. **A Regulação dos Setores de Infra-Estrutura no Brasil**. Rio de Janeiro. BNDES, 1999. Disponível em:
<www.bndes.gov.br/conhecimento/livro/eco90_07.pdf>. Acesso em: abril de 2007.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo : Makron, 1995. 1056 p.

RUMBAUGH, James et al. **Modelagem e projetos baseados em objetos**. 8 ed. Rio de Janeiro : Campus, 1994. 652 p.

QUEIRÓZ, Roosevelt B. O papel das agências reguladoras, **Gazeta Mercantil RS** 01/08/2002, pg. 2.

SANTANA, Angela. A reforma do Estado no Brasil : estratégias e resultados. VII **Congreso Internacional del CLAD sobre la Reforma del Estado y de la Administración Pública**, Lisboa, Portugal, Oct. 2002. Disponível em: <http://unpan1.un.org/intradoc/groups/public/documents/CLAD/clad0043328.pdf>. Acesso em: abril 2007.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. Tradução José Carlos Barbosa dos Santos. 4. Ed. Porto Alegre: Bookman, 2000.

ROXO, Letícia Figueiredo. **A Credibilidade das Reformas: Uma Análise do Setor Elétrico Brasileiro**. Rio de Janeiro, 2005. Dissertação (Mestrado em Ciências Econômicas) – Instituto de Economia, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2005. Disponível em: <http://www.gee.ie.ufrj.br/publicacoes/pdf/2005_cred_reformas.pdf>. Acesso em: abril de 2007.

WATER INDUSTRY COMMISSION FOR SCOTLAND - WICS. **Our work in regulating the Scottish water industry: Setting out a clear framework for the Strategic Review of Charges 2006-10**. Volume 1, july 2004. Disponível em: <www.watercommissioner.co.uk>. Acesso em: abril 2007.

WATER SERVICES REGULATION AUTHORITY – OFWAT. Information **for Regulation**. Volume 1, may 1995. Disponível em: <http://www.ofwat.gov.uk>. Acesso em: abril 2007.

WIKIPEDIA. A Enciclopédia Livre. ActiveX. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/ActiveX>. Acesso em: julho 2007.

WIKIPEDIA A Enciclopédia Livre. Ambiente de Desenvolvimento Integrado. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/Ambiente_de_desenvolvimento_integrado>. Acesso em: julho 2007.

WIKIPEDIA. A Enciclopédia Livre. DLL. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/DLL>. Acesso em: julho 2007.

WIKIPÉDIA. A Enciclopédia Livre. Engenharia de Software. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: maio 2007.

WIKIPEDIA. A Enciclopédia Livre. Orientação a objeto. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objeto>. Acesso em: maio 2007.

WIKIPEDIA. A Enciclopédia Livre. RAD. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/RAD>. Acesso em: julho 2007.

WIKIPEDIA. A Enciclopédia Livre. Shell. Wikimedia Foundation, Inc. US. Disponível em: <http://pt.wikipedia.org/wiki/Shell>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. BASIC. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/BASIC_programming_language>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. Event-driven programming. Wikimedia Foundation, Inc. US. Disponível em:

<http://en.wikipedia.org/wiki/Event_driven_programming_language>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. Graphical user interface. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/Graphical_user_interface>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. .NET Framework. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/.NET_Framework>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. Scripting language. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/Scripting_language>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. Visual Basic. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/Visual_Basic>. Acesso em: julho 2007.

WIKIPEDIA. The Free Encyclopedia. Visual Basic .NET. Wikimedia Foundation, Inc. US. Disponível em: <http://en.wikipedia.org/wiki/Visual_Basic_.NET>. Acesso em: julho 2007.

WORLD BANK. **Annotated reading list for a body of knowledge on the regulation of utility infrastructure and services**. October, 2004. Disponível em: <http://www.regulationbodyofknowledge.org>. Acesso em: abril de 2007.

ANEXO

Formulários do SOA – Sistema de Ouvidoria da Agergs:

Figura 5: Formulário Login
Fonte: Sistema de Ouvidoria da Agergs

Solicitação de Ouvidoria

Código Tipo Via Data / /

Área Responsável

Concessionária:

Usuário: Nome

Endereço

Bairro Cidade

CEP - Telefone ( ) - Celular ( ) -

E-mail

Descrição

Andamento

Situação Final

Arquivamento Arquivado Data / / SPI - . / -

Situação Prazo / /

Novo Salvar Excluir Pesquisar Cancelar Sair

Figura 6: Formulário de Cadastro
Fonte: Sistema de Ouvidoria da Agergs

Figura 7: Formulário de Pesquisa
Fonte: Sistema de Ouvidoria da Agergs

Figura 8: Formulário de Pesquisa de Usuários
Fonte: Sistema de Ouvidoria da Agergs

Solicitações de Ouvidoria - Relatórios

Data Inicial __/__/____

Data Final / /

Área de Atuação

- por Área de Atuação
- por Tipo/Assunto
- por Concessionária
- por Tipo/Assunto por Concessionária
- por Modo de Recebimento

Relatório | Sair

Figura 9: Formulário de Relatórios
Fonte: Sistema de Ouvidoria da Agergs