UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Impacto da Orientação por Objetos no Projeto de Software de Boa Qualidade

por

Alexander Thiago de Assis Oliveira Carvalho

Projeto Orientado em Computação I

Profa. Mariza A. S. Bigonha
Orientadora

Belo Horizonte, agosto de 2003

Índice

1. VISÃO GERAL. 3
1.1. INTRODUÇÃO. 3
1.2. PROBLEMA A SER PESQUISADO. 4
1.3. ORGANIZAÇÃO DO TEXTO. 4

2. PARADIGMA DA ORIENTAÇÃO POR OBJETOS E A ENGENHARIA DE SOFTWARE. 5
2.1. O QUE É ORIENTAÇÃO POR OBJETOS. 5
2.2. O QUE É A ENGENHARIA DE SOFTWARE. 7
2.3. BENEFÍCIOS DA PROGRAMAÇÃO ORIENTADA POR OBJETOS NO DESENVOLVIMENTO DE SOFTWARE. 9
2.4. **CONCLUSÃO.** 12

3. AVALIAÇÃO DO IMPACTO DA PROGRAMAÇÃO ORIENTADO POR OBJETOS NA QUALIDADE DO SOFTWARE. 13
3.1. INTRODUÇÃO. 13
3.2. LINGUAGENS AVALIADAS. 13
3.2.1. Smalltalk. *14*
3.2.2. Eiffel. *14*
3.2.3. C++. *15*
3.2.4. Java. *16*
3.3. DEFINIÇÕES IMPORTANTES. 17
3.4. RECURSOS AVALIADOS. 18
3.5. CONCLUSÃO. 18

4. CONJUNTO DE MÉTRICAS PARA O PARADIGMA DA ORIENTAÇÃO POR OBJETOS. 19
4.1. INTRODUÇÃO. 19
4.2. BASE TEÓRICA. 20
4.3. DEFINIÇÕES. 22
4.4. CONCLUSÃO. 26

5. CONCLUSÃO. 27

6. BIBLIORAFIA. 28

1. VISÃO GERAL.

1.1. Introdução.

Desde seu surgimento, a Engenharia de Software busca a forma mais eficaz de se produzir software com um alto grau de qualidade. Recentemente, seu melhor aliado nesta busca tem sido a programação orientada por objetos, a qual oferece diversos recursos que complementam e atingem os princípios da Engenharia de Software.

Com o passar dos anos as linguagens de programação começaram a ser projetadas ou reformuladas com uma visão voltada para a Orientação por Objetos (OO), o que tem como conseqüência, nos dias atuais, uma enorme gama de linguagens OO. Tal diversificação fez com que os projetistas de software utilizassem qualquer linguagem OO, pensando que pelo fato de uma linguagem ser orientação por objetos, isto seria suficiente para que a tornasse a melhor opção no desenvolvimento de um software. Como conseqüência, observa-se o uso indiscriminado das linguagens existentes, ignorando o tipo de aplicação a se desenvolver e qual a melhor linguagem OO para o desenvolvimento de tal aplicação. Por tais motivos este projeto tem como principal objetivo estabelecer critérios que identifiquem as várias linguagens OO e o impacto destas linguagens no desenvolvimento de Software de qualidade. Para atingir o objetivo proposto serão estudadas e analisadas um conjunto de métricas que podem ser utilizadas para comparar as várias linguagens existentes, seus pontos fortes e fracos e suas influências na produção de Softwares de qualidade.

O trabalho completo é composto de duas partes. A primeira delas se refere ao POC I e trata de escolher as linguagens de programação e estabelecer um conjunto de métricas para a avaliação dessas linguagens. O resultado final da pesquisa será apresentado na disciplina Projeto Orientado em Computação II.

1.2. Problema a ser Pesquisado.

Dados os recursos que serão tratados nesse trabalho é preciso encontrar uma forma de quantificar esses recursos e quantificar a sua influência na produção de software, uma vez que será preciso, de uma certa forma, comparar as linguagens avaliadas e verificar quando determinada linguagem deve ser aplicada.

Para isto, para cada um dos recursos descritos lhe será aplicado uma métrica para ver como ele se comporta diante da métrica e estabelecer um relação que possa ser medida e portanto dizer quando tal recurso está em um estado positivo ou negativo. O estado positivo indica benefício para a produção de software de qualidade e o negativo caso contrário. Após feita a análise de todas as métricas serão verificadas quais características associadas a cada linguagem fazem com que ela beneficie ou não de forma negativa ou positiva a qualidade do software, e desta forma definir características que faz com que determinada linguagem seja preferida para desenvolver determinada aplicação.

Este trabalho fica responsável por determinar as métricas e a base teórica que as compõem, ficando a parte de análise para a segunda parte do projeto.

1.3. Organização do Texto.

Este texto foi organizado da seguinte forma: a seção 2 trata de explicar o que é a orientação por objetos e Engenharia de Software, alem de mostrar como ambas estão ligadas e o que uma espera ou oferece a outra. A seção 3 trata das linguagens OO que foram escolhidas e de alguns conceitos que serão importantes para o entendimento do trabalho, além de tratar dos recursos que serão avaliados nas linguagens. A seção 4 é dedicada a explicar a base teórica que está por trás das métricas escolhidas e apresentar tais métricas. A seção 6 trata da conclusão da primeira parte deste projeto e a seção 7 refere-se a bibliografia utilizada.

2. PARADIGMA DA ORIENTAÇÃO POR OBJETOS E A ENGENHARIA DE SOFTWARE.

2.1. O que é Orientação por Objetos.

Um Sistema de Software deve fornecer respostas a questões do mundo exterior, interagir com o mundo exterior e criar novas entidades no mundo exterior, logo percebe-se que um Sistema de Software é um **Modelo Operacional** baseado em i*nterpretações* do mundo, cujos objetos que compõem o software devem ser representações dos objetos relevantes que constituem o mundo exterior. Baseado neste conceito é que a filosofia da orientação por objetos se impõe.

O conceito de Programação Orientada por Objetos já existe a um certo tempo. No final dos anos 60, a linguagem Simula67 [***], desenvolvida na Noruega, introduzia conceitos que hoje podem ser encontrados nas atuais linguagens orientadas por objetos. Em meados dos anos 70, o Centro de Pesquisa da Xerox (PARC) desenvolveu a Smalltalk[***], a primeira linguagem Orientada por Objetos totalmente pura. No início dos anos 80, a AT&T lança a Linguagem C++[***], uma evolução da linguagem C orientada por objetos.

O paradigma da orientação por objetos tem como principais objetivos reduzir a complexidade no desenvolvimento de software, aumentando sua produtividade e qualidade. A orientação por objetos não tem a intenção de substituir a programação estruturada tradicional, considere que a Orientação por Objetos nada mais é do que uma evolução de práticas que são recomendadas na programação estruturada, mas não formalizadas, como o uso de variáveis locais, visibilidade, escopo, etc. O modelo de objetos permite a criação de bibliotecas que tornam efetivos o compartilhamento e a reutilização de código, reduzindo o tempo de desenvolvimento e, principalmente, simplificando o processo de manutenção das aplicações, permitindo um maior grau de qualidade.

A dificuldade que existe em entender o paradigma da OO não está na forma como ele aborda o problema a ser solucionado, mas sim na mudança de paradigma em si. Enquanto os paradigmas imperativo, lógico e funcional têm como foco os procedimentos e funções, a orientação por objeto preocupa-se com os objetos e o relacionamentos entre eles. Além do conceito de objetos, a programação OO traz consigo novos conceitos tais como: classe,

herança, encapsulação, passagem de mensagem e polimorfismo, os quais são de suma importância para a produção de um software de qualidade.

A figura 1 apresenta uma analogia dos elementos da programação estruturada com os elementos da orientação por objetos.

Programação Estruturada	Programação Orientada a Objetos
Procedimentos e funções	Métodos
Variáveis	Instâncias de variáveis
Chamadas a procedimentos e funções	Mensagens
Tipos de dados definidos pelo usuário	Classes
	Herança
	Polimorfismo

Figura 1: Analogia entre a programação OO e a programação estruturada

Um objeto pode ser considerado como uma abstração de software que representa algo do mundo externo e que é formado por um conjunto de propriedades, variáveis, e métodos, procedimentos e funções. Um exemplo de objeto, *pessoa*, este objeto possui atributos como nome, idade, sexo, peso, altura, etc, e possui métodos que podem representar as ações de comer, andar, falar, etc.

Pode-se dizer que a orientação por objetos é uma coleção estruturada de implementações de tipos abstratos dados. Baseado nos conceitos de objetos, classes, encapsulação, herança, passagem de mensagem e polimorfismo, o paradigma da OO representa uma forma evolucionária de pensar e desenvolver Software, trazendo inúmeros benefícios à criação de programas, dentre os quais o mais notável é a reutilização de código, que reduz drasticamente os tempos de desenvolvimento e manutenção de programas.

2.2. O que é a Engenharia de Software.

A Engenharia de Software surgiu em meados dos anos 70 para tentar contornar a crise do software e dar um tratamento de *Engenharia* à produção de Software*,* ou seja, fazer de forma mais sistemática e controlada o desenvolvimento de sistemas de Software complexos [B97]. Um sistema de software complexo se caracteriza por um conjunto de componentes abstratos que são estruturas de dados e algoritmos encapsulados na forma de procedimentos, funções, módulos, objetos ou agentes, e interconectados entre si, compondo a arquitetura do software, que deverão ser executados em sistemas computacionais [B97].

Os fundamentos científicos para a Engenharia de Software envolvem o uso de modelos abstratos e precisos que permitem ao engenheiro especificar, projetar, implementar e manter sistemas de software, avaliando e garantido suas qualidades. Além disto, a Engenharia de Software deve oferecer mecanismos para se planejar e gerenciar o processo de desenvolvimento.

Em seu dicionário, Aurélio define a *Engenharia* como sendo:"*Arte de aplicar os conhecimentos científicos à invenção, aperfeiçoamento ou utilização da técnica industrial em todas as suas determinações"*. Esta definição mostra em suas entrelinhas que a *Engenharia* busca a qualidade, logo a Engenharia de Software busca a qualidade do software.

Vale ressaltar que além de ferramentas, princípios e elementos, todos voltados para a implementação do software, os quais fazem parte do escopo deste trabalho, a Engenharia de Software, como dito anteriormente, oferece mecanismos para se planejar e gerenciar o processo de desenvolvimento do software, mas estes mecanismos estão fora do tratamento deste projeto.

A qualidade do software buscada pela Engenharia de Software pode ser melhor caracterizada como uma combinação de diversos fatores, *externos e internos*, que juntos alcançam um alto grau de qualidade para os sistemas que são desenvolvidos utilizando estes fatores [B97].

Os fatores externos são aqueles que sua presença ou ausência em um produto de Software podem ser detectados pelos seus usuários, tais como velocidade ou facilidade de uso.

Os fatores internos são fatores que são perceptíveis somente pelos profissionais que estão envolvidos na produção do software, tais como modularidade e legibilidade.

Segundo Bertrand Meyer [B97], somente os fatores externos são os que realmente importam, mas estes fatores só podem ser alcançados via fatores internos.

A seguir segue uma possível lista de fatores externos de um software:

- **Ser Correto**: A capacidade que um software possui de ser correto está na habilidade de executar exatamente o que lhe foi pedido e definido por sua especificação.
- **Robustez**: Capacidade do software de reagir apropriadamente a condições anormais de execução.
- **Extensibilidade:** Facilidade de adaptar os produtos de software a mudanças na sua especificação inicial.
- **Reusabilidade:** Capacidade de utilizar-se dos mesmos elementos de software na construção de várias aplicações diferentes.
- **Compatibilidade:** Capacidade de combinar elementos do software com outros elementos de sistemas diferentes.
- **Eficiência:** Habilidade do software de usar da melhor forma possível os recursos de hardware, tais como tempo de processador, espaço ocupado nas memórias internas e externas e banda de passagem no comunicação com dispositivos.

De todos os fatores externos de qualidade os mais importantes são: **Correção, Robustez, Reusabilidade e Extensibilidade**. Esses fatores estão ligados diretamente a dois fatores internos importantes. A correção e robustez podem ser alcançados via fator **Confiabilidade**, já a Reusabilidade e Extensibilidade podem ser alcançados via fator **Modularidade.**

A Extensibilidade está ligada diretamente com os custos de manutenção do software, logo, a Engenharia de Software ajuda nos custos relativo ao desenvolvimento de um sistema de Software. O gráfico da figura 2 mostra a porcentagem de gastos com manutenção devido a diversos fatores, e o fator de maior impacto é relacionado às mudanças nos requisitos feitos pelos usuários, ou seja a

extensibilidade.

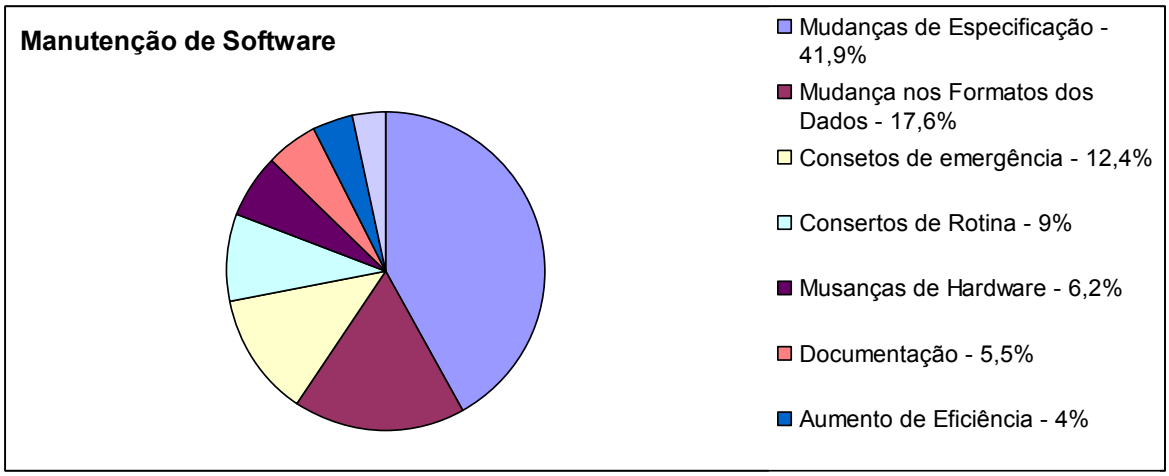

Figura 2: Gráfico de Manutenção em Software.
Fonte: Lintz, B.P. & Swanson, E.B., *Software Maintenance: A User/ Management Tug of War*, Data Management, pp. 26-30, april 1979.

2.3. Benefícios da Programação Orientada por Objetos no Desenvolvimento de Software.

Agora será mostrado porque a Orientação por Objetos oferece maiores benefícios para a produção de softwares de qualidade do que os outros paradigmas. Para isso será evidenciado os conceitos envolvidos na OO e quais os benefícios que estes conceitos traz para a Engenharia de Software no quesito qualidade.

Os conceito mais importantes da OO para alcançar a qualidade são: classe, herança, passagem de mensagem, polimorfismo e encapsulação.

As classes são recursos da OO que consistem em uma estrutura de dados possivelmente encapsulada, um conjunto de operações para manipular esta estrutura e uma interface bem definida. As classes provêem mecanismos que dão suporte à herança, ao polimorfismo e à encapsulação. Uma classe possui vários elementos, tais como campos e métodos, os métodos são as funções e procedimentos que fornecem o serviço de uma classe, já os seus campos são sua variáveis. Quando uma classe herda de outra, significa que a classe herdeira passa a ter os mesmos campos e métodos da classe herdada além dos seus próprios elementos, dando suporte ao fator reúso.

Os controladores de visibilidade são recursos que juntamente com as classes permitem a encapsulação, ou seja, implementam tipos abstratos de dados. É possível que a classe estabeleça o que vai ser mostrado ao mundo externo e o que diz respeito somente a ela, desta forma ela exporta seus serviços e esconde como estes serviços são implementados, dando suporte também ao fator reusabilidade, além do mais, permite a proteção dos seus dados, uma vez que objetos fora da classe não podem acessar o que não for permitido, logo entra em jogo o fator confiabilidade.

O polimorfismo aparece como sendo a capacidade que uma função possui de atuar da mesma forma sobre objetos de diferentes tipos, sendo um recurso muito poderoso na OO.

Pelo que foi explicado nos parágrafos anteriores, as classes oferecem vários recursos que permitem de forma bem explícita atingir a reusabilidade, mas o que é mais importante ainda é que esses recursos são internos ao desenvolvimento do software, logo como é possível atingir um fator externo de qualidade como o reúso, sendo que fatores externos só podem ser atingidos vias fatores internos? A explicação está no fator modularidade. Todos os conceitos existentes dentro da OO estão fortemente ligados ao fator modularidade e confiabilidade, sendo estes últimos os fatores internos mais importantes.

A programação modular traz consigo o conceito de módulo, os quais são "pedaços" de software organizados e auto contidos, logo podem ser compilados separadamente. Ao explicitar os critérios, regras e afins, que fazem um software ser modular, fica mais fácil para o leitor perceber o amplo conceito que está por traz da modularidade, ao invés de simplesmente definir um conceito sintético e portanto incompleto.

Para um método de construção de software ser dito modular, é preciso que ele siga cinco critérios fundamentais, os quais estão escritos abaixo:

- **"Decomposibilidade":** Um método de construção de software possui decomposibilidade se ele permite que o problema a ser resolvido possa ser decomposto em um número pequeno de subproblemas menos complexos, conectados por uma estrutura simples, e independente o suficiente para permitir que eles trabalhem separadamente uns dos outros.

- **“Composibilidade”:** Um método de construção de software possui composibilidade se ele favorece a produção de elementos de software, os quais podem ser livremente combinados uns com os outros para produzirem novos sistemas, possivelmente em um ambiente um pouco diferente de onde eles foram inicialmente produzidos.
- **“Entendibilidade”:** Um método de construção de software possui entendibilidade, se ele ajuda a produzir softwares que permitem ao leitor entender cada módulo do sistema sem ter que conhecer os outros, ou no pior caso, examinar apenas uma pequena parte de outros módulos.
- **Continuidade:** Um método de construção de software possui continuidade se uma mudança na especificação do software trouxer mudanças em somente um módulo ou em uma pequena parte deles.
- **Proteção:** Um método de construção de software possui proteção se uma condição anormal ocorrer em tempo de execução do módulo e ela tiver efeito somente sobre aquele módulo ou propagar somente para um pequeno grupo de módulos vizinhos.

Além destes critérios a modularidade estabelece as seguintes regras:

- **Mapeamento direto:** a estrutura modular proposta no processo de construção do sistema de software deve permanecer compatível com qualquer estrutura modular presente no processo de modelagem do domínio do problema.
- **Poucas Interfaces:** Todo módulo deve se comunicar com um número pequeno de outros módulos.
- **Interfaces Pequenas:** Se dois módulos se comunicam, eles devem trocar o mínimo de informação possível.
- **Interfaces Explicitas:** O local no código onde o módulo se comunica com os demais deve ser explícita e de fácil entendimento.
- **Ocultamento de Informação:** O módulo deve deixar os seus clientes terem acesso só no que for permitido.

Por última a modularidade pode ser classificada em cinco princípios:

- **O princípio das Unidades Modulares Lingüísticas:** Os módulos devem corresponder a unidades sintáticas na linguagem usada.
- **O princípio da auto documentação:** O desenvolvedor de um módulo deve tentar colocar toda a informação sobre o módulo dentro do seu próprio código.
- **Princípio do acesso uniforme:** Todos os serviços oferecidos pelo módulo devem ser disponíveis por uma notação uniforme, a qual não deve divulgar por exemplo se o módulo é implementado via de armazenamento ou via computação.
- **Princípio aberto-fechado:** Os módulos devem ser aberto para poderem ser alterados, mas devem estar fechados quando estiverem sendo usados.

Com todas as regras, critérios e princípios citados acima é possível perceber de uma forma mais clara o que está por traz do conceito de modularidade e como os elementos existentes na Orientação por Objeto estão ligados a este conceito.

Por fim resta o fator confiabilidade para ser tratado. Para um software ser confiável, ele deve ser robusto e correto. A robustez é proporcionada pela OO por meio dos vários recursos oferecidos por ela, por exemplo o tratamento de exceções. Além disso se um módulo for reutilizado significa que ele já foi usado outras vezes e a possibilidade de haver erros é muito pequena, pois ele já foi testado.

2.4. Conclusão.

Nesta seção foi apresentado os principais elementos envolvidos neste trabalho a Orientação por Objeto e a Engenharia de Software, e também como os conceitos existentes dentro de cada um estão relacionados entre si. O propósito da Engenharia de Software é encontrar caminhos para se construir softwares de qualidade, esta qualidade é vista como um conjunto de fatores externos e internos ao desenvolvimento do Software.

Os fatores internos mais importantes que devem ser considerados é a reusabilidade e a modularidade, os quais irão permitir alcançar os fatores externos correção, robustez, reusabilidade e extensibilidade. A chave para alcançar tais fatores está no conceito de modularidade, na escolha da estrutura de módulos, os quais irão permitir que o software seja reutilizado, aumentando a confiabilidade. Mas é preciso utilizar-se das ferramentas

corretas no momento de construir o software para que todos esses objetivos sejam alcançados. Essa ferramenta é a Orientação por Objetos. A orientação por objetos é a metodologia para construir software de grande porte modularmente.

3. AVALIAÇÃO DO IMPACTO DA PROGRAMAÇÃO ORIENTADO POR OBJETOS NA QUALIDADE DO SOFTWARE.

3.1. Introdução.

A primeira linguagem a incorporar facilidades para definir classes de objetos genéricos na forma de uma hierarquia de classes e sub-classes, foi a linguagem Simula, a qual foi idealizada em 1966, na Noruega, como uma extensão da linguagem ALGOL 60. As idéias de Simula serviram de base para as propostas de utilização de Tipos Abstratos de Dados, e também para Smalltalk.

3.2. Linguagens avaliadas.

Atualmente existem diversas linguagens orientadas por objetos puras ou que agregaram funcionalidades da OO. Para este trabalho foram escolhidas quatro linguagens OO para serem avaliadas. Como as linguagens escolhidas abrangem de maneira bem extensa e diversificada o universo da orientação por objeto, não foi preciso avaliar um número grande de linguagens, para ser capaz de tais avaliações.

As quatro linguagens escolhidas foram: Smalltalk, Eiffel, C++ e Java. Smalltalk foi escolhida por ter sido a primeira linguagem puramente orientada por objetos, e por isso mantém características bem expressivas do paradigma em questão. Eiffel foi escolhida para ser avaliada porque é uma linguagem elegante, possui um mecanismo de exceção muito bom e também devido ao fato de ser fácil de encontrar na literatura referências e comparações com outras. Já C++ e Java, por serem linguagens excessivamente utilizadas, foram as outras duas escolhas. Procurou-se também linguagens com diversidade de recursos.

As próximas subseções apresentarão um breve histórico sobre cada linguagem.

3.2.1. Smalltalk.

Smalltalk foi desenvolvida no Centro de Pesquisas da Xerox durante a década de 70 e incorporou, além das idéias de Simula-67, um outro conceito importante, graças a Alan Kay, um de seus idealizadores: o princípio de objetos ativos, os quais reagem a mensagens que ativam estados específicos do objeto. Ou seja, os objetos em Smalltalk deixam de ser meros dados manipulados por programas, e passam a ser vistos como processadores idealizados, individuais e independentes, aos quais podem ser transmitidos comandos em forma de mensagens.

Smalltalk, assim como outras linguagens orientadas por objetos, tem sido usada em aplicações variadas onde a ênfase está na simulação de modelos de sistemas, como automação de escritórios, animação gráfica, informática educativa, instrumentos virtuais, editores de texto e bancos de dados genéricos. Tais aplicações diferem substancialmente daquelas em que a ênfase está na resolução de problemas via algoritmos, tais como problemas de busca, otimização e resolução numérica de equações. Para essas aplicações, é mais adequado o uso de linguagens do paradigma Imperativo.

Uma Segunda versão do Smalltalk, o Smalltalk/V, foi desenvolvida pela Digitalk em Los Angeles, Califórnia, com fundadores da Ollivetti. Antes de ser comprada pela ParcPlace Systems Inc., a Digitalk era a líder de vendas deste produto.

A IBM desenvolveu o VisualAge Smalltalk em colaboração com a Object Technology International Inc. Atualmente, ObjectShare, antiga ParcPlace-Digitalk, e a IBM continuam sendo os principais distribuidores do ambiente de desenvolvimento Smalltalk.

Os principais recursos de Smalltalk são: a grande hierarquia de classes, que constitui a base para a maioria dos programas dela, e o poderoso ambiente de janelas em que seus programas são desenvolvidos.

3.2.2. Eiffel.

A linguagem Eiffel é uma avançada linguagem de programação orientada a objetos, que enfatiza o projeto e a construção de “software” reutilizável de alta qualidade. A linguagem Eiffel não corresponde a um sub-conjunto ou extensão de nenhuma outra linguagem. É uma linguagem que enfatiza o uso de técnicas de programação orientada a

objeto, além de permitir um interfaceamento com outras linguagens, tais como o C ou o C++.

A linguagem Eiffel suporta o conceito de projeto por contrato, para garantir a consistência do software gerado. A linguagem Eiffel foi criada por Bertrand Meyer, e desenvolvida por sua companhia, a "Interactive Software Engineering (ISE)". Seu projeto foi inspirado por algumas preocupações levantadas por engenheiros de software, diante da criação de sistemas mais complexos. Como linguagem, vem evoluindo continuamente, desde sua concepção, em 1986. Compiladores para a linguagem Eiffel podem ser encontrados para diversos sistemas operacionais e plataformas de hardware, destacando-se as seguintes: PC: DOS, OS/2, Windows 3.1, Windows 95, Windows NT, PC Unix (Interactive, SCO, and ESIX), Nextstep, LinuxOutras plataformas de Hardware: Sparc (SunOS & Solaris), NeXTStep, HP9000, DEC 5xxx, Sony News, DG Aviion, DEC Alpha OSF-1, DEC OpenVMS, RS6000, Pyramid, QNX, Silicon Graphics, Macintosh (Motorola & PowerPC). [G97]

Dentre outros recursos existentes em Eiffel destacam-se a herança, a vinculação dinâmica e o tratamento de exceções.

3.2.3. C++.

O C++ é uma linguagem de programação de propósito geral, desenvolvida por Bjarne Stroustrup, nos laboratórios da AT&T Bell, no início dos anos 80, como uma evolução do C, incorporando, dentre outras, as seguintes extensões: suporte para a criação e uso de tipos de dados abstratos, suporte ao paradigma de programação orientada a objeto, além de diversas outras pequenas melhorias nas construções existentes no C. Algumas de suas características são o uso de tipos estáticos, a definição de classes, funções virtuais e sobrecarga de operadores para o suporte à programação orientada a objeto, o uso de templates para programação genérica, além de prover facilidades de programação de baixo nível a exemplo do C.

Depois de seu "release" inicial pela AT&T, em 1985, diversas implementações do C++ tornaram-se disponíveis, em mais de 24 tipos de sistemas, desde PCs até mainframes. Em 1989, o número de usuários e diferentes implementações do C++ demandaram a geração de uma norma, sem a qual seria inevitável o surgimento de diversos dialetos. Em

1995 os comitês de normalização da ANSI e ISO C++ chegaram a um nível de estabilidade na linguagem. Basicamente, o C++ corresponde à linguagem descrita em [S91]. A norma que regula o C++ está atualmente em fase de "draft", sendo catalogada como X3J16/95-0087 pela ANSI e WG21/NO687 pela ISO. [G97]

3.2.4. Java.

A linguagem Java foi desenvolvida a partir de 1990 pela Sun Microsystems, como uma linguagem que pudesse executar o mesmo programa em múltiplas plataformas de hardware e software. Seu uso imediato era a execução de programas pela Internet. Para tanto, não poderia haver nenhum vínculo da linguagem com o hardware e/ou o sistema operacional utilizado, de modo que em princípio qualquer computador conectado à Internet e capaz de entender a linguagem, fosse capaz de executar seus programas. Outra especificação básica da linguagem diz respeito à segurança, ou seja, como as máquinas executando os programas em Java estariam em princípio executando programas sem garantias de confiabilidade e procedência, um programa em Java não poderia de modo algum influir na execução de outros programas e o próprio sistema operacional. Com isso, alguns recursos como alocação dinâmica de memória e acesso a arquivos foram sistematicamente eliminados.

A linguagem Java foi desenvolvida a partir de um subconjunto do C++, gerando uma linguagem que originalmente foi chamada de Oak. Em 1995, a linguagem foi re-batizada como Java e introduzida na comunidade Internet.

Java é uma linguagem parcialmente compilada e parcialmente interpretada. Um compilador Java transforma o programa fonte, escrito em Java, em arquivos-objeto chamados *bytecodes*. Esses *bytecodes* precisam ser executados então por interpretadores Java que são desenvolvidos para cada plataforma de hardware/software. Os *bytecodes* podem ser basicamente de dois tipos. O primeiro tipo tem acesso completo à máquina, ou seja, é capaz de manipular a memória, o console e o sistema de arquivos. Programas desse tipo são chamados de **aplicações** Java. O segundo tipo de *bytecode* possui uma série de restrições quanto ao acesso de memória, console e sistema de arquivos. Essas restrições são colocadas em nome da segurança, visto que seu destino é a elaboração de programas que serão distribuídos pela Internet, e por isso não provém de fonte conhecida ou confiável.

Esses *bytecodes* são chamados de Java **applets**. Os applets têm uma capacidade de atuação bem restrita, de modo que não possam causar danos ao sistema durante sua execução. Normalmente sua atuação se restringe a animações e interações mínimas com o usuário, sendo que seu uso é marcadamente em *browsers* do WWW (World Wide Web). Para executar applets, os browsers necessitam interpretar os bytecodes Java. Browsers com essa capacidade são chamados de *Java aware*. Uma outra variante da linguagem Java é o JavaScript, que são programas colocados em forma de código fonte, incluídos nos textos das páginas HTML. Programas escritos com o JavaScript não precisam ser compilados, nem geram bytecodes, sendo interpretados diretamente pelos browsers quando a página HTML é interpretada. [G97]

3.3. Definições Importantes.

A figura 3 apresenta algumas definições consideradas importantes para o entendimento deste trabalho. Muito embora algumas delas já tenham sido dadas de forma mais completa nas seções anteriores, vale a pena ressaltá-las pela sua importancia no que diz respeito às métricas abordadas na seção 4.

Atributo	Define as propriedades estruturais da classe, único dentro da classe e geralmente conhecido.
Classe	Um conjunto de elementos que compartilham uma estrutura comum e um comportamento que é manipulado por um conjunto de métodos. Esse conjunto de elementos serve como um modelo para a instanciação de novos objetos.
Coesão	O grau com que um método dentro de uma classe está relacionado com outro método dentro da mesma classe.
Acoplamento	Um objeto A está acoplado com um objeto B se e somente se A envia mensagem para B.
Herança	Uma relação entre classes, onde um objeto em uma classe adquire características de um ou mais objetos de outra classe.
Instanciação	Processo de criação de um objeto baseado em um modelo (classe) e relacionar ou adicionar dados específicos a estes objetos.
Mensagem	Pedido que um objeto faz a outro objeto para que este último faça determinada ação.
Método	Uma operação sobre um objeto, definida dentro da declaração da classe.
Objeto	Uma instanciação de alguma classe, a qual está apta a salvar um estado (informação) e a qual oferece um número de operações para examinar ou afetar este estado.

Operação	Uma ação efetuada por ou sobre um objeto, disponível para todas as instâncias da classe.

Figura 3: Principais definições associadas às linguagens OO.

Fonte: Definições tiradas de [R98]

3.4. Recursos Avaliados.

Os recursos da orientação por objetos que serão avaliados são aqueles apresentados na seção 2, tais como: classe, herança, passagem de mensagem, encapsulação e polimorfismo. Para todos os recursos citados, exceto polimorfismo, o conjunto de métricas escolhidas servirá para avaliá-los. Polimorfismo deve ser avaliado separadamente.

3.5. Conclusão.

Nesta seção foram apresentados um breve histórico das linguagens Smalltalk, Eiffel, C++ e Java, escolhidas para serem avaliadas com o conjunto de métricas apresentado na seção 4. Estas métricas vão avaliar alguns dos recursos dessas linguagens como por exemplo: classe, herança e encapsulação. O polimorfismo será avaliado separadamente, uma vez que não possui relação direta com os conceitos citados anteriormente.

Dentro do que será avaliado, os conceitos apresentados serão de suma importância para o entendimento do processo avaliativo das linguagens e o impacto que estas possuem na produção de Softwares de qualidade.

4. CONJUNTO DE MÉTRICAS PARA O PARADIGMA DA ORIENTAÇÃO POR OBJETOS.

4.1. Introdução.

Na literatura existem várias métricas com diferentes focos e crenças. Muitas delas são difíceis de serem verificadas e analisadas, pois (1) podem depender muito da tecnologia que está sendo avaliada, (2) apresentam um alto grau de dificuldade em coletar dados ou (3) podem ser muito gerais.

Pelos motivos citados anteriormente, foram escolhidas as métricas da ontologia de Bunge [B77], [B79]. Fazem parte deste conjunto nove métricas. Dessas métricas, três foram excluídas por aplicarem a qualquer paradigma de programação e portanto não fazem parte do escopo deste trabalho, o qual destina-se a avaliação de linguagens Orientadas por Objetos.

Essas métricas servirão de base para avaliar as quatro linguagens OO escolhidas e a influência dessas linguagens na Engenharia de Software. As métricas apresentadas utilizam dos conceitos classe, herança, encapsulação e passagem de mensagem para que possam ser aplicadas.

Com base nos recursos que serão avaliados e nos conceitos apresentados na seção 2 [C94] foram escolhidas as seis métricas citadas abaixo:

- **Número de Métodos por Classe.**
- **Profundidade na Árvore de Herança.**
- **Número de Filhos.**
- **Acoplamento entre as Classes.**
- **Resposta de uma Classe**
- **Falta de Coesão nos Métodos.**

Com este conjunto e métricas as linguagens escolhidas serão avaliadas, identificado quais as características que cada linguagem possui que beneficia ou prejudica cada elemento que está sendo tratado em cada métrica.

4.2. Base Teórica.

Existem várias metodologias para o desenvolvimento de softwares no paradigma da orientação por objetos. O escolhido para estudo foi proposto por Booch [B91], por possuir os pontos essenciais para o desenvolvimento de softwares OO.

Esta metodologia apresenta quatro passos principais:

1. *Identificação das classes e objetos:* Neste passo elementos do problema são analisados e identificados como potenciais classes ou objetos.
2. *Identificar a semântica das classes e dos objetos:* Neste passo é estabelecido o significado de cada classe e objeto identificado no passo anterior.
3. *Identificação da relação entre as classes e objetos:* Neste passo são identificadas as relações existentes entre os objetos e classes. Herança e visibilidade (encapsulação) são conceitos importantes nesta fase.
4. *Implementação das classes e objetos:* Esta é a fase em que implementa-se os detalhes internos, os métodos e seus comportamentos são definidos aqui.

As classes são elementos centrais da metodologia de Booch. Devido a este fato as métricas que serão tratadas neste projeto são todas desenvovlidas para avaliar a complexidade de se projetar classes.

Um projeto baseado em orientação por objetos pode ser visto como um sistema relacional, o qual é constituído por um conjunto de elementos, classes e objetos, um conjunto de relações e um conjunto de operações binárias [R79]. Sobre esses objetos uma notação formal seria:

D = (**A**, R_1...Rn, O_1...On), onde D representa o projeto OO, A o conjunto de elementos (classes e objetos), R_1...Rn são as relações empíricas sobre os elementos de A

(maior que, menor que, etc) e O_1...On são as operações binárias sobre os elementos (combinações).

O conceito de complexidade é a melhor forma para entender o que é uma relação empírica entre os objetos e classes. Por exemplo, se uma classe possui mais métodos do que outra pode-se dizer que tal classe é mais complexa do que a outra. Este tipo de idéia, intuição, é denominada de *ponto de vista*. O *ponto de vista*, é uma maneira intuitiva de análise e portanto deve ser considerado como o ponto de início para a definição das métricas.

Um ponto de vista pode ser visto como uma relação binária ≥ definida sobre o conjunto **P** (conjunto de todas as possibilidades de projeto), para P, P', P'' Є **P**. Dados os axiomas:

$$\begin{cases} P \geq P' \text{ ou } P' \geq P \text{ (Completeza: P é mais complexo que P' ou P' é mais complexo que P) .} \\ P \geq P', P' \geq P'' \rightarrow P \geq P'' \text{ (Transitividade: Se P é mais complexo que P' e P' é mais complexo que P'', então P é mais complexo que P'').} \end{cases}$$

Para ser possível efetuar uma medida sobre um projeto de objetos, o sistema de relacionamento empírico como definido precisa ser transformado em um sistema relacional formal. Por enquanto, seja um sistema relacional formal **F** definido como:

F = (**C**, S_1...Sn, B_1...Bn), onde C é um conjunto de elementos (números reais), S1...Sn são relações formais sobre os elementos de C (<,>,=) e B_1...Bn são operações binárias sobre os elementos de C (+,-,*).

Isto requer que a transformação seja feita por uma métrica μ, a qual mapeia um sistema empírico **D** em um sistema formal **F**. Para cada elemento a Є **D**, μ (a) Є **F**. Deve ser notado que μ preserva e não altera a noção implícita que possui as relações empíricas. O

exemplo da figura 4 mostra o mapeamento de um sistema de relacional empírico em um sistema relacional formal [A91]:

Sistema de Relacionamento Empírico	Sistema de Relacionamento Formal
Altura das crianças da escola	Números reais
Relações: igual, mais alto que. Criança P é mais alta que a criança P'	Relações: = ou > 36 pés > 30 pés
Operações binárias: Combinações: duas crianças em pé uma sobre a outra.	Operação binária: +: some os dois números reais associados às duas crianças.

Figura 4: Relacionamento Empírico X Relacionamento formal.

Embora o exemplo desta figura seja simples, a idéia é mostrar que a representação numérica produzida pela transformação em um sistema formal, ajuda a entender melhor o sistema empírico. Além disso, mesmo tendo sido um exemplo simples, a transformação pode provar ser valiosa no entendimento da complexidade de software onde as relações de complexidade não estão muito visíveis ou não muito bem explicadas [A91].

4.3. Definições.

Consistente com a ontologia aqui tratada [B77], objetos são definidos independentes das considerações de implementação e abrangência das noções de encapsulação, independência e herança. De acordo com esta ontologia, o mundo é visto como sendo composto de objetos, que são referidos como indivíduos substanciais, e conceitos. A noção chave é que indivíduos substanciais possuem propriedades. A propriedade é uma característica que um indivíduo substancial possui através de herança. Um observador pode atribuir características aos indivíduos, mas estes são atributos e não propriedades. As propriedades não existem sozinhas, elas precisam ser anexadas aos indivíduos para que possam fazer sentido. Um indivíduo e sua coleção de propriedades, juntos, constituem um objeto. Um objeto **X** pode ser representado da seguinte maneira:

X = (x, p(x)), onde x é o indivíduo substancial e p(x) é a coleção finita de propriedades deste indivíduo.

x pode ser um token ou um nome pelo qual o objeto é representado dentro do sistema. Na terminologia da orientação por objetos, as variáveis de instância com os seus métodos são as propriedades do objeto [B87].

Embora já tenham sido apresentados de forma mais simples alguns dos conceitos que serão tratados a seguir, a partir de agora eles serão apresentados de uma maneira mais formal.

Acoplamento, em termos ontológicos é definido como: "dois objetos são acoplados se e somente se pelo menos um deles atua sobre o outro, X é dito atuar sobre Y se a história de Y é afetada por X, onde história é definida como os estados ordenados cronologicamente que algo possui ao longo do tempo" [W84].

Deixe X = (x, p(x)) e Y = (y, p(y)) serem dois objetos.

p(x) = {Mx} U {Ix}

p(y) = {My} U {Iy}

onde {Mi} é o conjunto dos métodos e {Ii} é o conjunto de variáveis e instância do objeto i.

Considerando a definição de acoplamento acima, temos que qualquer ação de {Mx} sobre {My} ou {Iy} significa que X e Y estão acoplados, isto vale também para qualquer ação de {My} sobre {Mx} ou {Ix}. Resumindo, quando métodos de uma classe usa métodos de outra classe ou suas variáveis, então as duas classes em questão estão acopladas.

Bunge define *Coesão* como sendo a similaridade σ( ) de dois elementos, ou seja, a interseção do conjunto de propriedades destes dois elementos [B77].

$\sigma(X,Y) = p(x) \cap p(y)$

Deve ficar claro que variáveis não soa propriedades de métodos, mas é consistente com a noção que métodos de um objeto estão intimamente conectados com suas variáveis.

$\sigma(M_1,M_2) = \{I_1\} \cap \{I_2\}$

onde $\sigma(M_1,M_2)$ é o grau de similaridade dos métodos M_1 e M_2 e {Ii} é o conjunto de variáveis de instância usadas pelo método Mi.

Ex: Considere $\{ I_1\} = \{a,b,c,d,e\}$e $\{ I_2\} = \{a,b,e\}$. $\{I_1\} \cap \{I_2\}$ não é vazio e $\sigma(M_1,M_2) = \{a,b,e\}$.

Se uma classe possui diferentes métodos que atuam em diferentes operações sobre as mesmas variáveis de instância, a classe é dita ser coesiva. Pode-se observar que coesão está intimamente ligada a encapsulação. Para uma classe ser encapsulada ela precisa ser coesa.

Um outro conceito que será tratado é o de *Complexidade de um Objeto*. Bunge define complexidade de um indivíduo como a "numerosidade de sua composição", logo um indivíduo complexo é aquele que possui um número grande de propriedades [B77]. Utilizando-se deste conceito considera-se que a complexidade de um objeto é a cardinalidade de seus conjuntos de propriedades. Complexidade de $(x,p(x)) = | p(x) |$, onde $| p(x) |$ é a cardinalidade de p(x).

Escopo de propriedades: Definindo de forma simples, uma classe é o conjunto de objetos que possuem propriedades comuns (métodos e variáveis de instância).

As classes podem ser organizadas de forma hierárquica, logo entra em cena o conceito de *herança*. A hierarquia de herança pode ser representada como uma árvore de classes com nodos, folhas e uma raiz. Com esta representação pode-se definir dois conceitos: *profundidade de herança*, o qual está relacionado com a posição da classe dentro da árvore de herança, logo a profundidade dentro da árvore, e o *número de filhos* de uma

classe, uma vez que os nodos de uma árvore de herança são constituídos de classes. A profundidade da árvore de herança indica qual o impacto que uma determinada classe irá sofrer devido as propriedades de seus antecessores, e o número de filhos irá indicar o impacto que as propriedades de determinada classe irão causar a seus descendentes.

A *passagem de mensagem* é outra definição que deve ser tratada. Na orientação por objetos, os objetos se comunicam, primeiramente, por meio de passagem de mensagem. Uma mensagem causa a um objeto um comportamento particular pela invocação de um método específico. Os métodos podem ser vistos como os responsáveis pelas respostas à mensagens recebidas. O conjunto de respostas de uma classe é o conjunto de todos os métodos que podem ser chamados em resposta a uma mensagem para um objeto da classe. Note que, devido à definição de acoplamento, métodos fora da classe também podem ser chamados, uma vez que métodos internos à classe podem chamar métodos externos. O conjunto de respostas será finito desde que as propriedades da classe sejam finitas e exista um número finito de classes.

Como última definição a ser tratada tem-se a *Combinação de classes*. Como já foi observado por alguns autores como Booch, o projeto de classes é um processo interativo que envolve subclasses, que são novas classes baseadas em outras já existentes, fatoração, separar classes existentes em classes menores, e composição ou combinação, as quais consiste unir classes existentes em uma única classe. A ontologia de Bunge provê a base para definir a combinação de classes pelo princípio da agregação aditiva de duas ou mais classes. A combinação de duas ou mais classes resulta em uma nova classe cujas propriedades são a união das propriedades das classes que a originaram.

Considere X = (x, p(x)) e Y = (y, p(y)) como sendo duas classes. Logo X + Y é definido como (z, p(z)), onde z é o token com o qual X + Y é representado e p(z) é dado por:

$$p(z) = p(x) \cup p(y)$$

Tem-se como o único resultado que pode ser definitivo da combinação de duas classes como sendo a eliminação de todas as mensagens anteriores entre as duas classes.

4.4. Conclusão.

Aqui foi apresentado os aspectos mais importantes dentro de um projeto orientado por objeto e foram apresentadas as definições que tais aspectos utilizam ou de alguma forma estão relacionados com elas. Com isso definiu-se um conjunto de métricas específicas para a orientação por objetos, as quais serão as escolhidas para alcançar os objetivos deste projeto.

5. CONCLUSÃO.

A Engenharia de Software prega a qualidade no desenvolvimento de novos sistemas, para isso ela estabelece um série de critérios e regras que devem ser seguidos e alcançados. A orientação por objetos entra neste cenário para poder dar o suporte necessário a este objetivo da ES, mas deve-se lembrar que não é correto utilizar a OO de forma indiscriminada, deve-se fazer uma avaliação da linguagem escolhida e a aplicação a ser desenvolvida. Para tanto estabeleceu-se os recursos que devem ser avaliados, tais como: classe, herança, encapsulação e passagem de mensagem, para poder fazer tal tipo de estudo.

Este trabalho teve como objetivo "parcial" estabelecer quais os conceitos e definições estão envolvidos com os recursos a serem avaliados e definir um conjunto de métricas a serem utilizadas para que seja possível "quantificar" as características que cada linguagem possui, podendo então compará-las.

Os objetivos foram caracterizados como parciais, pois o objetivo final deste trabalho, o qual foi dividido em duas partes, é estabelecer alguns critérios para evitar o uso indiscriminado da Orientação por Objetos dentro da Engenharia de Software, mostrando os pontos fortes e fracos de cada linguagem que será analisada.

6. BIBLIORAFIA.

[A91] A. A. Kaposi, "*Measurement Theory",* in Software Engineer's Ref. Book, J. McDermind, Ed., Oxford: Butterworth-Heinemann Ltd., 1991.

[B77] M. Bunge, *Treatise on Basic Philosophy: Ontology I: The Furniture of the World.* Boston: Rieldel, 1977.

[B77] M. Bunge, *Treatise on Basic Philosophy: Ontology I: The World of Systems.* Boston: Rieldel, 1979.

[B87] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, and N, Ballou, *"Data model issues for object oriented applications,"* ACM Trans. Office Inform. Syst., vol. 5, pp. 3-26, 1987.

[B91] G. Booch, *Object Oriented Design with Applications.* Redwood City, CA: Benjamin/Cummings, 1991.

[B97] Meyer, Bertrand. *Object-oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 2nd Edition, 1254 pág., 1997.

[B03] Bigonha, Mariza Andrade da Silva; BIGONHA, Roberto da Silva. *Programação Orientado por Objetos*, transparências de aula. Universidade Federal de Minas Gerais, departamento de Ciência da Computação. Belo Horizonte, 2003.

[B03] BIGONHA, Mariza Andrade da Silva; BIGONHA, Roberto da Silva. *Linguagens de Programação*, transparências de aula. Universidade Federal de Minas Gerais, departamento de Ciência da Computação. Belo Horizonte, 2003.

[C94] Chidamber, Shyam and Kemerer, Chris, "*A Metrics Suite for Object Oriented Design*", IEEE Transactions on Software Eng, June, 1994, pp. 476-492.

[F03] Filho, Wilson de Pádua Paula. *Engenharia de Software Fundamentos, Métodos e Padrões.* São Paulo, 2ª edição, 2003.

[G97] Gudwin, Ricardo R., "*Notas de aula para a disciplina EA877*", Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e Computação, 1997. Disponível em: < http://www.aerohobby.hpg.ig.com.br/Hobbies/1/ling_prog.pdf>.

[P84] R. E. Prather, "An axiomatic theory of software complexity measures." *Comput. J.*, vol. 27, pp 340-346. 1984.

[R79] F. Roberts, *Encyclopedia of Mathematical and its Applications.* Reading, MA: Addison-Wesley, 1979.

[R98] Rosenberg, Linda H, *Applying and Interpreting Object Oriented Metrics.* Utah, April 1998. Disponível em: http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html

[S91] Stroustrup, Bjarne , *"The C++ Programming Language"* , ed. Addison Wesley 2ª edição,1991.

[S99] Sebesta, Robert W., *Linguagens de Programação*, Trad. José Carlos Barbosa dos Santos, 2000. Bookman, 4a. edição, 1999.

[W84] I. Vessey and R. Weber, "*Research on structured programming: An empiricist's evaluation,*" IEEE Transactions on Software Eng., vol. SE-10, pp 394-407, 1984.

[W88] E. Weyuker, “Evaluating software complexity measures.”, *IEEE Transactions Software Engineering.,* vol. 14, pp 1357-1365, 1988.

[W90] Watt, David. *Programming Language Concepts and Paradigms*, C.A.R. Hoare series editor, Prentice Hall International Series in Computer Science, 1ª edição. 1990.