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

Relatório Final
Projeto Orientado em Computação II

Impacto da Orientação por Objetos em Projeto de Software de Boa Qualidade

por

Alexander Thiago de Assis Oliveira Carvalho

alexande@dcc.ufmg.br

Orientadora

Profa. Mariza A. S. Bigonha

mariza@dcc.ufmg.br

Belo Horizonte, Janeiro de 2004

Sumário

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 orientada por objetos, isto seria suficiente para que a tornasse a melhor opção no desenvolvimento de um software. Neste cenário, observa-se o uso indiscriminado das linguagens existentes, sem a preocupação com o tipo de aplicação a se desenvolver e qual a melhor linguagem OO para o desenvolvimento de tal aplicação. Por tais motivos este projeto teve como principal objetivo estudar e analisar um conjunto de métricas que pudessem ser utilizadas para comparar as várias linguagens existentes, seus pontos fortes e fracos e assim estabelecer critérios que identifiquem as várias linguagens OO e o impacto destas linguagens no desenvolvimento de software de qualidade.

Dentre os recursos disponíveis nas linguagens orientadas por objetos foram avaliados : classe, herança, passagem de mensagem, encapsulação e polimorfismo que são tratados no Capítulo 2. Para todos os recursos citados, exceto polimorfismo, o conjunto de métricas escolhidas serviu para avaliá-los. Polimorfismo foi avaliado separadamente.

1.2. Problema Pesquisado.

Dados os recursos escolhidos para cada LOO: classe, herança, passagem de mensagem, encapsulação e polimorfismo, inicialmente, foi preciso encontrar uma forma de quantificá-los e quantificar a sua influência na produção de software, uma vez que era preciso, de uma certa forma, comparar as linguagens avaliadas e verificar quando determinada linguagem podia ser aplicada.

Para isto, para cada um dos recursos definidos lhe foi aplicado uma métrica para ver como ele se comportava diante da métrica escolhida e estabelecer assim uma relação que pudesse ser medida e portanto dizer quando tal recurso poderia ser considerado em um estado positivo ou negativo. O estado positivo indica benefício para a produção de software de qualidade e o negativo caso contrário. Depois de feita a análise de todas as métricas, foram verificadas quais as características associadas a cada linguagem que faziam com que ela beneficiasse de forma positiva ou negativa na qualidade do software. Este resultado foi muito importante, pois a partir dele é possível identificar características que fazem com que determinada linguagem seja preferida para desenvolver determinada aplicação.

1.3. Organização do Texto.

Este texto foi organizado da seguinte forma: o Capítulo 2 explica o que é a orientação por objetos e Engenharia de Software, além de mostrar como ambas estão ligadas e o que uma espera ou oferece a outra. O Capítulo 3 apresenta as linguagens OO que foram escolhidas e alguns conceitos que são importantes para o entendimento do trabalho, além de tratar dos recursos que serão avaliados nas linguagens. O Capítulo 4 expõe a base teórica que está por trás das métricas escolhidas e apresenta tais métricas. Também apresenta o critério de avaliação que foi utilizado para analisar o conjunto de métricas e explica de forma detalhada cada uma das seis métricas, mostrando seus conceitos e características. O Capítulo 5 trata de mostrar a análise feita de cada uma das quatro linguagens em relação ao conjunto de métricas, evidenciando os pontos positivos e negativos de cada linguagem. O Capítulo 6 refere-se à conclusão deste projeto e o Capítulo 7 refere-se a bibliografia utilizada. O Apêndice A apresenta um benchmark que possui como objetivo mostrar as características referentes à qualidade e legibilidade do software.

2. Paradigma da Orientação por Objetos e a Engenharia de Software.

2.1. Introdução.

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 *interpretações* do mundo, cujos objetos que compõem o software devem ser representações dos objetos relevantes que constituem o mundo exterior [B97]. Baseado neste conceito é que a filosofia da orientação por objetos se impõe.

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.

Este capítulo expõe os conceitos mais importantes sobre orientação por objetos bem como seus aspectos e características. São descritos também o que é a engenharia de software e o que ela prega quanto a qualidade do software, além de mostrar os recursos da OO que beneficiam a produção de software de qualidade.

2.2. O que é Orientação por Objetos.

O conceito de Programação Orientada por Objetos já existe a um certo tempo. No final dos anos 60, a linguagem Simula67 [MM], 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[MM], a primeira linguagem Orientada por Objetos totalmente pura. No início dos anos 80, a AT&T lança a Linguagem C++[MM], uma evolução da linguagem C orientada por objetos.

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.

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. Por exemplo, considere o 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.

2.3. 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 devem ser executados em sistemas computacionais [B97].

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 escopo 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 robustez 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], 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, extraídos de [B97], de um software de acordo com sua importância:

- **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 na comunicação com dispositivos.

Dentre os fatores externos de qualidade mais importantes: **Correção, Robustez, Reusabilidade e Extensibilidade**, os fatores Reusabilidade e Extensibilidade estão relacionados ao conceito de orientação por objeto. Os demais se aplicam a qualquer paradigma de L.P. 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 está relacionado às mudanças nos requisitos feitos pelos usuários, ou seja a extensibilidade.

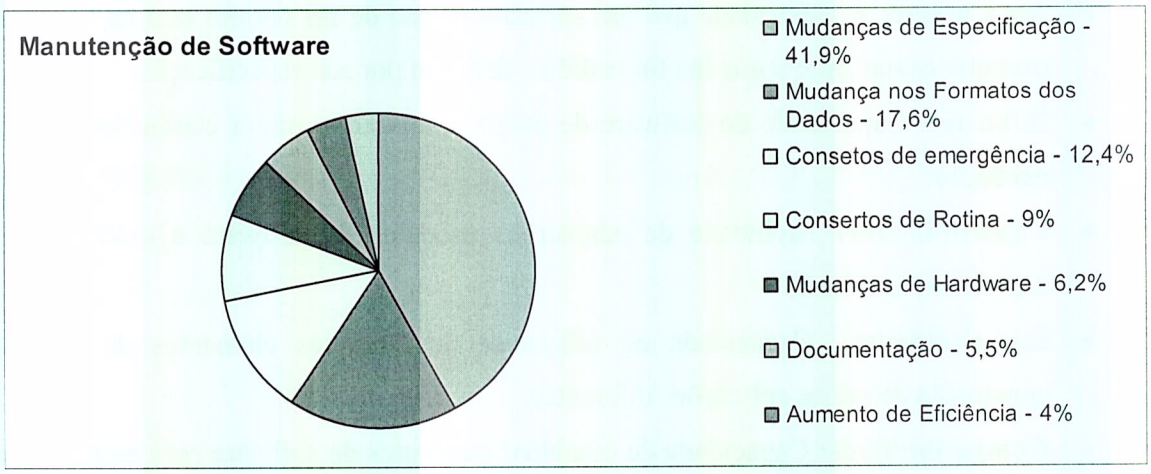

Figura 2: Gráfico de Manutenção em Software.
Fonte: Lintz, B.P. & Swanson, E.B., *Software Maintenance: A User/ Management Tug of War*, Data Management, pp. 26-30, april 1979.

2.4. Recursos oferecidos pela Programação Orientada por Objetos.

Para mostrar porque a orientação por objetos oferece maiores benefícios na produção de software de qualidade que os outros paradigmas, são evidenciados os conceitos envolvidos na OO e os benefícios que estes conceitos traz para a Engenharia de Software no quesito qualidade.

Os fatores internos mais importantes para o desenvolvimento de sistemas OO são a modularidade e a legibilidade. Estes fatores são alcançados via aplicação de conceito da OO como: 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 suas 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 reuso.

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 reuso, sendo que fatores externos só podem ser atingidos vias fatores internos? A explicação está no fator modularidade.

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 foram extraídos de [B97] e estão apresentados a seguir:

- **"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, segundo [B97]:

- **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 último a modularidade pode ser classificada em cinco princípios, segundo [B97]:

- **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 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.

2.5. Conclusão.

Neste capítulo foram apresentados os principais elementos envolvidos no paradigma Orientado por Objeto e na Engenharia de Software, e também como os conceitos existentes dentro de cada um deles estão relacionados entre si. O propósito da Engenharia de Software é encontrar caminhos para se construir software 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 são 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. Linguagens Orientadas por Objeto.

3.1. Introdução.

Atualmente existem diversas linguagens orientadas por objetos puras ou que agregaram funcionalidades da OO. Como o objetivo desta pesquisa é avaliar o impacto da POO na qualidade do software escolheu-se quatro linguagens OO para serem avaliadas: Smalltalk [SM], Eiffel [EF], C++ [FC] e Java [SMS]. Como essas linguagens 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.

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 linguagens. Já C++ e Java, por serem linguagens excessivamente utilizadas. Procurou-se também linguagens com diversidade de recursos. Além disso todas as quatro linguagens possuem os recursos de classe, herança, encapsulação, passagem de mensagem e polimorfismo, recursos fixados para a análise em questão.

As próximas subseções apresentam um breve histórico sobre cada linguagem.

3.2. Smalltalk.

Smalltalk [SMS] 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.

Além dos recursos citados na Seção 7.2 Smalltalk possui: grande hierarquia de classes, que constitui a base para a maioria de seus programas, e o poderoso ambiente de janelas em que seus programas são desenvolvidos.

3.3. Eiffel.

A linguagem Eiffel [EF] 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]

Além dos recursos citados na Seção 3.1 Eiffel destaca-se pela vinculação dinâmica e o tratamento de exceções que possui.

O suporte à programação Orientada por Objetos em Eiffel é similar ao de Java. Em nenhum dos casos é suportada a programação baseada em procedimentos e em ambos os casos quase todas as vinculações de mensagens são dinâmicas. Contudo, a elegância e o projeto limpo das classes e da herança de Eiffel estão em segundo lugar somente em relação ao da Smalltalk. Eiffel aceita herança múltipla [S99].

3.4. C++.

O C++ [TC] 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 e de possuir os recursos citados na Seção 3.1.

Depois de seu lançamento 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]

As classes em C++ podem possuir uma classe pai ou não, diferente de Smalltalk e Java. Em termos de controle de acesso, a herança em C++ é mais confusa do que nas demais linguagens avaliadas [S99].

3.5. Java.

A linguagem Java [SMS] 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]. Além de todos os recursos citados acima Java possui aqueles citados na Seção 3.1.

Java suporta diretamente somente herança simples, porém, ela inclui outro recurso que proporciona uma versão de herança múltipla, a Interface. O encapsulamento em Java pode ser construídos de duas formas, através das classes ou através dos pacotes, um pacote é um encapsulamento lógico. Java não suporta classes sem pais [S99].

3.6. Definições Importantes.

A Figura 3 apresenta algumas definições consideradas importantes para o entendimento deste trabalho. Muito embora algumas delas já tenham sido definidas de forma mais completa no Capítulo 2, vale a pena ressaltá-las pela sua importância no que diz respeito às métricas abordadas no Capítulo 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]

3.7. Conclusão.

Neste capítulo foram apresentados um breve histórico das linguagens Smalltalk, Eiffel, C++ e Java, escolhidas para serem avaliadas com o conjunto de métricas apresentado no Capítulo 4. Estas métricas avaliam alguns dos recursos dessas linguagens como por exemplo: classe, herança e encapsulação. O polimorfismo é avaliado separadamente, uma vez que não possui relação direta com os conceitos citados anteriormente.

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 pelas seguintes razões:

- podem depender muito da tecnologia que está sendo avaliada.
- apresentam um alto grau de dificuldade em coletar dados.
- podem ser muito gerais.

Pelos motivos citados anteriormente, foram escolhidas as métricas da ontologia de Bunge [B77], [B79]. Este conjunto é composto de nove métricas, três delas não serão consideradas por aplicarem a qualquer paradigma de programação e portanto não fazem parte do escopo deste trabalho, as demais são:

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

Essas métricas, detalhadas na Seção 4.4, servirão de base para avaliar as quatro linguagens OO escolhidas, Smalltalk, Eiffel, C++ e Java, e a influência dessas linguagens no desenvolvimento de software. As métricas apresentadas utilizam dos recursos de classe, herança, encapsulação e passagem de mensagem para que possam ser aplicadas.

A Seção 4.2 trata de introduzir a base teórica que dá suporte às métricas escolhidas. Já a Seção 4.3 diz respeito aos conceitos que serão utilizados, tratando-os de maneira mais precisa.

4.2. Base Teórica.

Existem várias metodologias para o desenvolvimento de software 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 [C94].

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 tratadas neste projeto são todas desenvolvidas para avaliar a complexidade de se projetar classes [C94].

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 representa 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) [C94].

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 [C94].

Um ponto de vista pode ser visto como uma relação binária $\geq$ definida sobre o conjunto **P**, conjunto de todas as possibilidades de projeto, para P, P', P'' $\in$ **P**. Dados os axiomas:

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

Para ser possível efetuar uma medida sobre objetos, o sistema de relacionamento empírico 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 (+,-,*)[C94].

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 $\in$ **D**, μ (a) $\in$ **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 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 via 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 [C94]. 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 tratados a seguir, a partir de agora eles são apresentados de uma forma mais precisa.

- *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.

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

  σ(X,Y) = p(x) ∩ p(y)

Deve ficar claro que variáveis não são 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.

Por exemplo 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 coesiva. Pode-se observar que coesão está intimamente ligada a encapsulação. Para uma classe ser encapsulada ela precisa ser coesa [C94].

- *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) [C94].
- *Escopo de propriedades:* Definindo de forma simples, uma classe é o conjunto de objetos que possuem propriedades comuns, métodos e variáveis de instância [C94].

  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 [C94].

- 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.
- *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 em 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) U 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. Critério para Avaliação das Métricas.

Alguns estudiosos da área [B79] recomendam que as métricas possuam algumas propriedades para que aumentem sua utilidade. Como exemplo temos Basili e Reiter [B79], os quais sugerem que as métricas devem ser sensitivas às diferenças externas observáveis no desenvolvimento e no ambiente, e devem corresponder a noções intuitivas sobre as diferentes características entre os diversos recursos que estão sendo medidos. A maioria das propriedades recomendadas são quantitativas, o que faz com que essas propriedades sejam informais nas suas avaliações das métricas [C94].

Para se ter um estudo mais rigoroso na avaliação das métricas propostas é desejável que se tenha um conjunto de critérios para avaliá-las. Neste capítulo são apresentadas as propriedades escolhidas para a avaliação do conjunto de métricas proposto. A Seção 4.4.1 apresenta de maneira detalhada cada propriedade. Já a seção 4.4.2 apresenta algumas suposições que são necessárias para o melhor entendimento das métricas.

4.4.1. Conjunto de Propriedades.

O conjunto de propriedades escolhidas foram definidas por Weyuker [W88]. Vale citar que Cherniavsky e Smith [C91] sugerem que estas propriedades devem ser usadas com cuidado, pois elas proporcionam condições necessárias, mas não suficientes para boas métricas de complexidade. Das nove propriedades de Weyuker, são apresentadas somente seis delas, pois três são atendidas por todas as métricas pertencentes ao conjunto escolhido, são elas granularidade, propriedade de renomeação e a sétima propriedade de Weyuker [W88]. A primeira diz que deve existir um conjunto finito de elementos com o mesmo valor de métrica. Como todas as métricas envolvem classes e as classes são um conjunto finito de elementos, ela sempre será satisfeita. Já a segunda propriedade diz que quando o nome de uma entidade

muda, a métrica deve permanecer a mesma. Como todas as métricas escolhidas não possuem interferência nos nomes das classes e métodos, todas as métricas satisfazem esta propriedade. Já a terceira propriedade diz que a permutação dos elementos dentro do item que está sendo medido pode alterar o valor da métrica. Este tipo de propriedade é mais aplicada a programação tradicional, onde existem recursos como o if-then-else que pode causar este tipo de efeito na métrica, logo não é uma propriedade que seja específica da OO, portanto não está sendo incluída no nosso conjunto. As seis propriedades restantes, as quais serão usadas para avaliar as métricas apresentadas na seção 4.1, estão descritas nas subseções seguintes [C94].

- ***Propriedade 1:*** *"Noncoarseness".*

  Dada uma classe P e uma métrica μ, uma outra classe Q pode ser sempre encontrada tal que: $\mu(P) \neq \mu(Q)$. Isto implica que nem toda classe pode ter o mesmo valor para uma métrica, caso contrário ela perde seu valor como uma medida.

- ***Propriedade 2:*** *Não unicidade (Noção de equivalência).*

  Pode existir classes distintas P e Q, tal que $\mu(P) = \mu(Q)$. Isto implica que duas classes podem ter o mesmo valor de métrica, ou seja, as duas classes são igualmente complexas.

- ***Propriedade 3:*** *Detalhes de Projeto são Importantes.*

  Dado dois projetos de classes, P e Q, os quais provêem a mesma funcionalidade, não implicando que $\mu(P) = \mu(Q)$. A especificação da classe deve influenciar no valor da métrica. A intuição por trás da propriedade 3 é que embora dois projetos de classes tenham a mesma função, os detalhes de projeto importam na determinação da métrica para a classe.

- ***Propriedade 4:*** *Monotonicidade.*

  Para todas as classes P e Q, o seguinte deve ser segurado:
  $\mu(P) \leq \mu(P + Q)$ e $\mu(Q) \geq \mu(P + Q)$ onde P + Q implica combinação de P e Q. Lembrando que $\mu(P) + \mu(Q) \neq P + Q$. Isto implica que a métrica para a combinação de

duas classes nunca ser menor do que a métrica para cada uma das classes componentes.

- ***Propriedade 5:*** *Não equivalência de Interação.*

Existem P, Q e R tal que: μ (P) = μ(Q) não implica que μ(P + R) + μ(Q + R). Isto sugere que a interação entre P e R pode ser diferente da interação entre Q e R, resultando em um valor complexidade para P + R e um para Q + R.

- ***Propriedade 6:*** *Interação aumenta Complexidade.*

Existem P e Q tal que: μ (P) + μ(Q) < μ(P + Q). O princípio por trás desta propriedade é que quando duas classes são combinadas, a interação entre as classes pode aumentar o valor da métrica de complexidade.

4.4.2. Suposições.

Algumas suposições básicas [C94] fazem referência à distribuição dos métodos e variáveis na discussão de cada uma das propriedades, essas suposições estão detalhadas a seguir:

- Suposição 1.

Seja:

X_i = Número de métodos de uma dada classe *i.*

Y_i = O número de métodos chamados de um dado método *i.*

Z_i = O número de variáveis usadas por um método *i.*

C_i = O número de acoplamentos entre uma dada classe *i* e todas as outras classes.

X_i, Y_i, Z_i, C_i são variáveis discretas randômicas cada uma caracterizada por alguma função de distribuição geral. Todos os X_i's são independentes e identicamente distribuídos (i.i.d.). O mesmo é verdade para as outras três variáveis. Isto sugere que o número de métodos, variáveis e acoplamentos seguem uma distribuição estatística que não é aparente para o observador do sistema. O observador não pode prognosticar as variáveis, métodos, etc. de uma classe baseado no conhecimento das variáveis, métodos e acoplamentos de uma outra classe do sistema.

- Suposição 2.

Duas classes podem possuir um número finito de métodos idênticos, de tal forma que a combinação dessas duas classes resulta em uma terceira com os mesmos métodos tornando redundante. Por exemplo, uma classe A possui uma método *"escreva"*, o qual escreve uma mensagem na tela; outra classe B possui o mesmo método *"escreva"*. Ao combinar as duas classes resultando a classe C, o projetista pode querer ter somente um método *"escreva"*, ao invés de ter dois métodos iguais.

- Suposição 3.

A árvore de herança está cheia, isto é, existe uma raiz, nodos intermediários e folhas. Esta suposição diz simplesmente que uma aplicação não consiste somente de classes isoladas, ou seja, existe o uso de subclasses.

4.5. As Métricas.

Na Seção 4.1 foi apresentado um conjunto de métricas utilizadas para avaliar e comparar as linguagens OO escolhidas, as próximas seções detalham cada uma dessas métricas bem como faz uma avaliação analítica [C94] de cada uma delas.

4.5.1. Métodos Ponderados Por Classe (MPC)

4.5.1.1. Definição.

Considere uma classe C_1, com métodos M_1 ,..., M_n que estão definidos na classe. Deixe c_1, ..., c_n ser a complexidade dos métodos *. Então:

$$\mathbf{MPC} = \sum_{i=1}^{n} Ci$$

Se todas as complexidades dos métodos são consideradas como sendo unidades, então MPC = n, onde n é o número de métodos.

4.5.1.2. Base Teórica.

MPC está relacionado diretamente com a definição dada por Bunge da complexidade de um elemento. Uma vez que os métodos são propriedades de classes e a complexidade é determinada pela cardinalidade de suas propriedades, portanto, o número de métodos de uma classe é a medida da definição da classe, bem como os atributos que a constitui, uma vez que atributos também fazem parte das propriedades de uma classe.

4.5.1.3. Pontos de Vista.

A definição de ponto de vista foi dada na Seção 4.2, a partir de agora este conceito é importante para o entendimento das métricas.

Na MPC os pontos de vistas a serem considerados são:

- O número de métodos e a complexidade dos métodos envolvidos é um indicador de quanto tempo e esforço são necessários para desenvolver e manter a classe.

- Quanto maior o número de métodos em uma classe maior será o potencial de impacto em seus filhos, classes herdeiras, uma vez que os filhos herdam todos os métodos definidos na classe pai.
- Classes que possuem uma número grande de métodos são específicas para uma determinada aplicação, dificultando o reuso.

4.5.1.4. Avaliação Analítica.

Da suposição 1, o número de métodos em uma classe P e outra classe Q são i.i.d., isto implica que existe um probabilidade diferente de zero que existe um Q tal que $\mu(P) \neq \mu(Q)$, conseqüentemente a *propriedade 1* é satisfeita. Similarmente, existe uma probabilidade diferente de zero que existe um R tal que $\mu(P) = \mu(R)$. Logo a *propriedade 2* é satisfeita. A escolha do número de métodos em uma classe é decisão de projeto e independe da funcionalidade da classe. Logo a *propriedade 3* é satisfeita. Deixe $\mu(P) = n_P$ e $\mu(Q) = n_Q$, então $\mu(P + Q) = n_P + n_Q - \delta$, onde δ é o número de métodos comuns entre P e

Q. É fácil de observar que o valor máximo de δ é o mín(n_P, n_q). Conseqüentemente, $\mu(P + Q) \geq n_P + n_Q$. mín(n_P, n_Q). Segue que $\mu(P + Q) \geq \mu(P)$ e $\mu(P + Q) \geq \mu(Q)$, satisfazendo assim, a *propriedade 4*. Agora, deixe que $\mu(P) = n$ e $\mu(Q) = n$, e exista uma classe R tal que tenha um número de métodos δ em comum com Q (conforme suposição 2) e métodos β em comum com P, onde $\delta \neq \beta$. Deixe $\mu(R) = r$;

então $\mu(P + R) = n + r - \beta$
então $\mu(Q + R) = n + r - \delta$

Logo, $\mu(P + R) \neq \mu(Q + R)$ e a *propriedade 5* é satisfeita. Para qualquer duas classes P e Q, $n_P + n_Q - \delta \leq n_P + n_Q$, ou seja, $\mu(P + Q) \leq \mu(P) + \mu(Q)$ para qualquer P e Q. Logo, a *propriedade 6* não é satisfeita.

---

* A complexidade não foi definida, pois desta forma permite uma aplicação mais geral desta métrica. Sendo uma decisão de implementação escolher o que será considerado como complexidade de um método.

4.5.2. Profundidade da Árvore de Herança (PAH)

4.5.2.1. Definição.

A profundidade da herança de uma classe é a métrica PAH para classe. Em casos onde apareçam herança múltipla, a PAH será a maior distância entre a raiz e o nodo.

4.5.2.2. Base Teórica.

A PAH está relacionada com a noção que Bunge possui sobre escopo de propriedades. PAH é uma medida de quantos antecessores podem afetar uma determinada classe.

4.5.2.3. Pontos de Vista.

Na PAH os pontos de vistas a serem considerados são:

- Quanto mais profunda uma classe é hierarquicamente, maior o número de métodos que esta classe pode herdar, fazendo com que seja mais complexo prever seu comportamento.
- Árvores profundas constituem maiores complexidades de projeto, uma vez que mais métodos e classes estão envolvidos.
- Quanto mais profunda uma classe é hierarquicamente, maior o potencial de reuso de métodos herdados.

4.5.2.4. Avaliação Analítica.

Pela suposição 3, a hierarquia de herança tem uma raiz e folhas. A profundidade de herança de uma folha é sempre maior que a da raiz. Logo a *propriedade 1* é satisfeita. Também, uma vez que cada árvore possui pelo menos algum nodo com irmãos, pela suposição 3, sempre existirá pelo menos duas classes com a mesma profundidade de herança, logo *propriedade 2* é satisfeita. O projeto de uma classe envolve escolher quais propriedades a classe deve herdar para exercer sua função. Em outras palavras, profundidade de herança depende do projeto, logo a *propriedade 3* é satisfeita.

Quando quaisquer duas classes P e Q são combinadas, existem três possíveis casos: No primeiro caso P e Q são irmãs, como mostram as Figuras 5 e 6. Neste caso, $\mu(P) = \mu(Q) = n$ e $\mu(P + Q) = n$, logo *propriedade 4* é satisfeita.

No segundo caso P e Q não são irmãs nem filhas uma da outra, como mostram as Figuras 7 e 8. Se P e Q são alocadas em uma posição imediatamente anterior a de B e C (posição de P) na árvore, a classe combinada não pode herdar métodos de X, embora se P + Q é alocada como um filho imediato de X (posição de Q), a classe combinada ainda pode herdar métodos de todos os antecessores de P e Q. Logo, P + Q serão alocados na posição de Q. Neste caso $\mu(P) = x$, , $\mu(Q) = y$ e $y > x$. $\mu(P + Q) = y$, logo $\mu(P + Q) > \mu(P)$ e $\mu(P + Q) > \mu(Q)$ e *propriedade 4* é satisfeita. Já no terceiro caso quando uma é filha de outra, como mostram as Figuras 9 e 10, $\mu(P) = n$, $\mu(Q) = n + 1$, mas $\mu(P + Q) = n$, ou seja, $\mu(P + Q) < \mu(Q)$. *Propriedade 4* não é satisfeita.

Deixe P e Q' serem irmãs, ou seja, $\mu(P) = \mu(Q') = n$, e deixe R ser uma filha de P. Então $\mu(P + R) = n$ e $\mu(Q' + R) = n + 1$. Logo, $\mu(P + R)$ não é igual a $\mu(Q' + R)$. Portanto a *propriedade 5* é satisfeita. Para quaisquer duas classes P e Q, $\mu(P + Q) = \mu(P)$ ou $= \mu(Q)$. Portanto, $\mu(P + Q) \leq \mu(P) + \mu(Q)$, logo, a *propriedade 6* não é satisfeita.

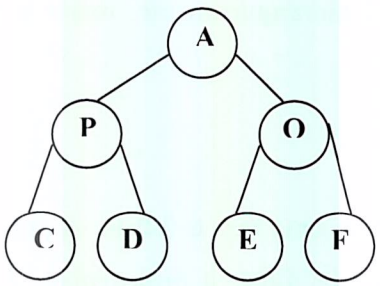

Figura 5.

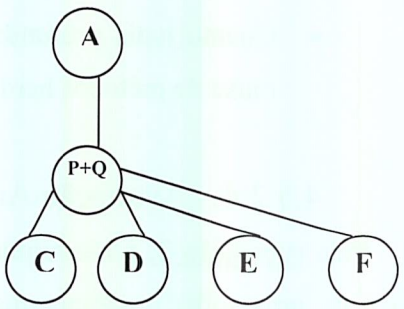

Figura 6.

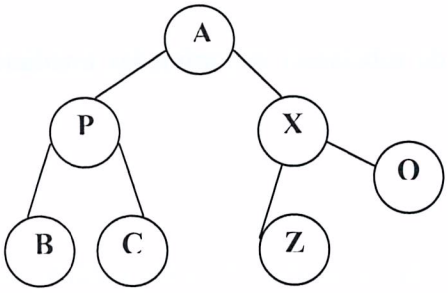

Figura 7.

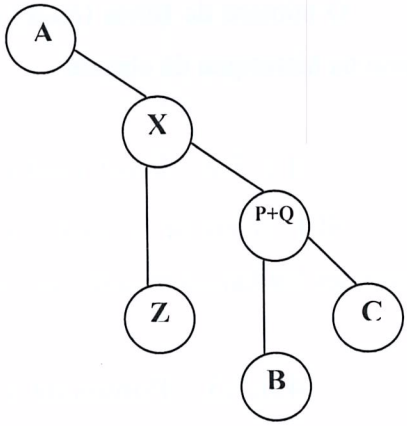

Figura 8.

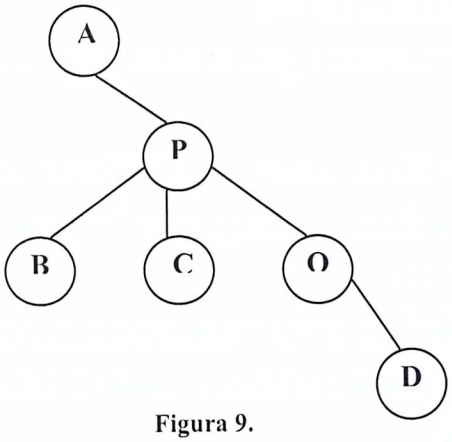

Figura 9.

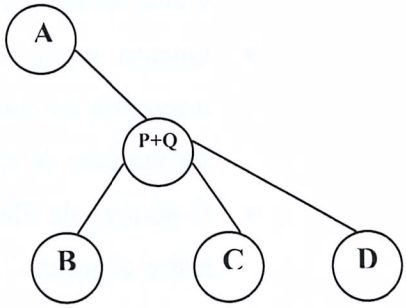

Figura 10.

4.5.3. Número de Filhos (NDF)

4.5.3.1. Definição.

O número de filhos (NDF) é o número de subclasses subordinadas imediatas de uma classe na hierarquia de classes.

4.5.3.2. Base Teórica.

NDF refere-se a noção de escopo de propriedades. É uma medida de quantas subclasses herdam os métodos de sua classe pai.

4.5.3.3. Pontos de Vista.

Na NDF os pontos de vistas a serem considerados são:

- Quanto maior o número de filhos, maior o grau de reuso, uma vez que herança é uma forma de reuso.
- Quanto maior o número de filhos, maior a probabilidade de abstração imprópria da classe pai. Se uma classe tem um grande número de filhos, pode ser um caso de abuso de subclasses.
- O número de filhos dá uma idéia da potencial influência que uma classe possui sobre o projeto. Se uma classe possui um número grande de filhos, ela deve requerer mais testes de métodos.

4.5.3.4. Avaliação Analítica.

Deixe P e R serem folhas, $\mu(P) = \mu(R) = 0$, deixe Q ser a raiz. $\mu(Q) > 0$. $\mu(P) \neq \mu(Q)$, logo a *propriedade 1* é satisfeita. Uma vez que $\mu(R) = \mu(P)$, a *propriedade 2* é satisfeita. O projeto de uma classe envolve decisões do escopo dos métodos declarados dentro da classe, ou seja, as subclasses da classe. O número de subclasses é portanto dependente do projeto de implementação da classe. Logo, a *propriedade 3* é satisfeita.

Deixe P e Q serem duas classes com n_P e n_Q sendo suas subclasses respectivamente, logo, $\mu(P) = n_P$ e $\mu(Q) = n_Q$. Combinando P e Q**, irá gerar uma única classe com $n_P + n_Q - \delta$ subclasses, onde δ é o número de filhos que P e Q possuem em comum. Claramente, δ é 0 se n_P ou n_Q é 0. Se Q é uma subclasse de P, então P + Q terá $n_P + n_Q - 1$ subclasses. Logo, em geral o número de subclasses de P + Q é $n_P + n_Q - \beta$, onde $\beta = 1$ ou δ. Agora, $n_P + n_Q - \beta \geq n_P$ e

$n_P + n_Q - \beta \geq n_Q$. Isto pode ser escrito como: $\mu(P + Q) \geq \mu(P)$ e $\mu(P + Q) \geq \mu(Q)$ para todo P e para todo Q. Logo a *propriedade 4* é satisfeita. Deixe que P e Q tenham n filhos cada um e R seja um filho de P que possui r filhos. $\mu(P) = n = \mu(Q)$. A classe obtida pela combinação de P e R terá $(n - 1) + r$ filhos, considerando uma classe obtida pela combinação de Q e R terá n +r filhos, o que significa que $\mu(P + R) \neq \mu(Q + R)$. Logo, a *propriedade 5* é satisfeita. Dadas duas classes P e Q com n_P e n_Q filhos respectivamente, segue a seguinte relação:

$$\mu(P) = n_P \text{ e } \mu(Q) = n_Q$$

$$\mu(P + Q) = n_P + n_Q - \delta$$

onde δ é o número de filhos comuns entre P e Q. Portanto, $\mu(P + Q) \leq \mu(P) + \mu(Q)$ para todo P e Q. Logo, a *propriedade 6* não é satisfeita.

4.5.4. Acoplamento entre Classes (AEC)

4.5.4.1. Definição.

AEC para uma classe é o número de classes com as quais ela está acoplada.

4.5.4.2. Base Teórica.

AEC está relacionada com a noção de que um objeto está acoplado com outro objeto se um deles atua sobre o outro, ou seja, métodos de um usa métodos ou variáveis de outro. Uma vez que objetos de uma mesma classe possuem as mesmas propriedades, duas classes são acopladas quando métodos declarados em uma classe usa métodos ou variáveis de outra classe.

4.5.4.3. Pontos de Vista.

Na AEC os pontos de vistas a serem considerados são:

- Acoplamento excessivo entre classes é prejudicial para o projeto modular e para o reuso. Quanto mais independente uma classe for mais fácil será reutilizá-la em outra aplicação.
- Para aumentar a modularidade e promover o encapsulamento, acoplamento entre classes deve ser mínimo. Quanto maior o número de acoplamento, maior a probabilidade de mudanças afetar outras partes do projeto, dificultando a manutenção.

- Uma medida de acoplamento é útil para determinar quão complexo é o teste da aplicação. Quanto mais acoplamento entre classes uma aplicação tenha, mais rigorosos deverão ser os testes.

4.5.4.4. Avaliação Analítica.

Pela suposição 1, existem classes P, Q e R tal que $\mu(P) \neq \mu(Q)$ e $\mu(P) \neq \mu(R)$, desta forma satisfazendo as *propriedades 1 e 2*. Acoplamento entre classes ocorre quando métodos de uma classe usa métodos ou variáveis de outra classe, logo como esse métodos e variáveis serão utilizados depende do projeto das classes e não da funcionalidade da classe. Logo a *propriedade 3* é satisfeita. Deixe P e Q serem quaisquer duas classes com $\mu(P) = n_P$ e $\mu(Q) = n_Q$. Se P e Q são combinadas, a classe resultante terá $n_P + n_Q - \delta$, onde δ é alguma função de métodos de P e Q. Claramente, $n_P - \delta > 0$ e $n_Q - \delta > 0$ uma vez que a redução de acoplamentos não pode ser maior do que o número original. Logo,

$$n_P + n_Q - \delta \geq n_P \text{ para todo P e Q e}$$
$$n_P + n_Q - \delta \geq n_Q \text{ para todo P e Q}$$

ou seja, $\mu(P + Q) \geq \mu(P)$ e $\mu(P + Q) \geq \mu(Q)$ para todo P e Q. Então a *propriedade 4* é satisfeita. Deixe P e Q serem duas classes quaisquer tal que $\mu(P) = \mu(Q) = n$, e deixe R ser uma outra classe com $\mu(R) = r$.

$$\mu(P + Q) = n + r - \delta, \text{ similarmente}$$
$$\mu(Q + R) = n + r - \beta.$$

Dado que δ e β são funções independentes, elas não serão iguais , ou seja, $\mu(P + R)$ não é igual a $\mu(Q + R)$, satisfazendo a *propriedade 5*. Para quaisquer duas classes P e Q, $\mu(P + Q) = n_P + n_Q - \delta$.

$$\mu(P + Q) = \mu(P) + \mu(Q) - \delta \text{ o qual implica que}$$
$$\mu(P + Q) \leq \mu(P) + \mu(Q) \text{ para todo P e Q.}$$

Logo a *propriedade 6* não é satisfeita.

4.5.5. Resposta de uma Classe (RDC)

4.5.5.1. Definição.

RDC = |RS| onde RS é o conjunto de respostas da classe.

4.5.5.2. Base Teórica.

O conjunto resposta da classe pode ser expressado como: $RS = \{M\} \cup_{\text{todo i}} \{R_i\}$ onde $\{R_i\}$ = conjunto dos métodos chamados pelo método i e {M} = conjunto de todos os métodos da classe.

O conjunto resposta de uma classe é o conjunto de métodos que podem potencialmente serem executados em resposta a uma mensagem recebida por um objeto da classe. A cardinalidade do conjunto é medida pelos atributos do objeto na classe. Uma vez que esses atributos incluem métodos chamados de fora da classe, ela é também uma medida da comunicação potencial entre a classe e outra classe.

4.5.5.3. Pontos de Vista.

Na RDC os pontos de vistas a serem considerados são:

- Se um grande número de métodos podem ser invocados em resposta a uma mensagem, o teste e depuração da classe torna-se mais complicado uma vez que requer um nível maior de entendimento da pessoa que irá testar.
- Quanto maior o número e métodos que podem ser invocados de uma classe, maior a complexidade da classe.
- O pior valor de resposta possível aparecerá nos testes.

4.5.5.4. Avaliação Analítica.

Seja:

X_p = RDC para a classe P

X_Q = RDC para a classe Q.

X_p e X_Q são funções do número de métodos e acoplamento externo de P e Q respectivamente. Segue da suposição 1 (uma vez que funções de variáveis randômicas *i.i.d.* são também *i.i.d.*) que X_p e X_Q são *i.i.d.* Logo, existe uma probabilidade diferente de erro que

existe um Q tal que $\mu(P) \neq \mu(Q)$ resultando que a *propriedade 1* é satisfeita. Também existe uma probabilidade diferente de zero de que existe um Q tal que $\mu(P) = \mu(Q)$, logo a *propriedade 2* é satisfeita. Uma vez que a escolha dos métodos é uma decisão de projeto, a *propriedade 3* é satisfeita. Seja P e Q duas classes com RDF de P = n_P e RDF de Q = n_Q. Se estas duas classes são combinadas para formar uma outra classes, a resposta para este classe criada dependerá se P e Q possuem algum método comum. Claramente, existem três casos possíveis: o primeiro será quando P e Q não possuem métodos comuns nem os seus métodos utilizam métodos comuns, logo a combinação de P + Q terá o conjunto resposta = $n_P + n_Q$. No segundo caso P e Q possuem métodos comuns, logo o conjunto resposta será menor que $n_P + n_Q$. E o último caso P e Q não possuem métodos em comum, mas os métodos de P e Q utilizam métodos comuns, logo o conjunto resposta será menor que $n_P + n_Q$. Nos segundo e terceiro casos, $\mu(P + Q) = n_P + n_Q - \delta$, onde δ é alguma função dos métodos de P e Q. Claramente, $n_P + n_Q - \delta \geq n_P$ e $n_P + n_Q - \delta \geq n_Q$ para todos os possíveis P e Q. Logo, a *propriedade 4* é satisfeita.

Seja P e Q duas classes tal que $\mu(P) = \mu(Q) = n$, e deixe R ser uma outra classe com $\mu(R) = r$.

$$\mu(P + R) = n + r - \delta, \text{ similarmente}$$
$$\mu(Q + R) = n + r - \beta.$$

Dado que δ e β são funções independentes, elas não serão necessariamente iguais, ou seja, $\mu(P + R)$ não é necessariamente igual a $\mu(Q + R)$, satisfazendo a *propriedade 5*. Para quaisquer duas classes P e Q,

$$\mu(P + Q) = \mu(P) + \mu(Q) - \delta \text{ o qual implica que}$$
$$\mu(P + Q) \leq \mu(P) + \mu(Q) \text{ para todo P e Q.}$$

Logo a *propriedade 6* não é satisfeita.

4.5.6. Falta de Coesão nos Métodos (FCM)

4.5.6.1. Definição.

Considere uma classe C_1 com n métodos $M_1,M_2...,M_n$. Seja $\{I_j\}$ = conjunto de variáveis usadas pelo método M_i.

Existe n tal que $\{I_1\},...,\{I_n\}$. Seja $P = \{(I_i,I_j) \mid I_i \cap I_j = \text{vazio}\}$ e $Q = \{(I_i,I_j) \mid I_i \cap I_j = \text{vazio}\}$. Se todo n os conjuntos $\{I_1\},...,\{I_n\}$ são vazios então seja P = vazio.

$$FCM = |P| - |Q|, \text{ se } |P| > |Q|$$

Se não* = 0.

Por exemplo considere uma classe C com três métodos M_1,M_2 e M_3. Seja $\{I_1\}$ = {a,b,c,d,e} e $\{I_2\}$ = {a,b,e} e $\{I_3\}$ = {x,y,z}. $\{I_1\} \cap \{I_2\}$ não é um conjunto vazio, mas $\{I_1\} \cap \{I_3\}$ e $\{I_2\} \cap \{I_3\}$ são conjuntos nulos. FCM é o número de interseções nulas – número de interseções não vazias, que neste caso é 1.

4.5.6.2. Base Teórica.

Esta métrica utiliza a noção de grau de similaridade de métodos. O grau de similaridade para dois métodos M_1 e M_2 na classe C_1 é dado por:

$\sigma( ) = \{I_1\} \cap \{I_2\}$ onde $\{I_1\}$ e $\{I_2\}$ são os conjuntos de variáveis usadas pelos métodos M_1 e M_2.

A FCM é o número de pares de métodos cuja similaridade é 0, ou seja, $\sigma( )$ é um conjunto nulo, menos o número de pares de métodos cuja similaridade não é zero. Quanto maior o número de métodos similares, mais coesa é a classe, a qual é consistente com noções tradicionais de coesão que mede a inter-relação entre partes de um programa. Se nenhum dos método de uma classe não usa variáveis, então os métodos não possuem similaridade e FCM para a classe será zero. O valor de FCM está intimamente ligado às variáveis e métodos da classe, logo é uma medida dos atributos dos objetos.

---

* Note que a métrica FCM para uma classe onde |P| = |Q| será zero. Isto não implica coesão máxima, uma vez que no conjunto das classes com FCM = 0, alguma pode ser mais coesiva que outras.

4.5.6.3. Pontos de Vista.

Na FCM os pontos de vistas a serem considerados são:

- Coesão de métodos em uma classe é desejável, desde que ela promova encapsulação.
- Falta de coesão implica que as classes devem provavelmente serem divididas em duas ou mais subclasses.
- Qualquer medida de desigualdade de métodos ajuda a identificar falhas no projeto das classes.
- Baixa coesão aumenta complexidade, aumentando a probabilidade de erros durante o processo de desenvolvimento.

4.5.6.4. Avaliação Analítica.

Seja

X_P = FCM para a classe P

X_Q = FCM para a classe Q.

X_P e X_Q são funções do número de métodos e de variáveis de P e Q respectivamente. Segue da suposição 1 (uma vez que funções de variáveis randômicas *i.i.d.* são também *i.i.d.*) que X_P e X_Q são *i.i.d.* Logo, existe uma probabilidade diferente de zero que existe um Q tal que $\mu(P) \neq \mu(Q)$, logo a *propriedade 1* é satisfeita. Também existe uma probabilidade diferente de zero de que existe um Q tal que $\mu(P) = \mu(Q)$, logo a *propriedade 2* é satisfeita. Uma vez que a escolha dos métodos e variáveis são decisões de projeto, a *propriedade 3* é satisfeita.

Suponha que a classe P tenha três métodos M_1, M_2, M_3 e M_2 e M_3 usam variáveis comuns, enquanto M_1 não possui variável comum com M_2 e M_3. A FCM para P será 1. Agora, seja Q uma outra classe com três métodos, todos usam variáveis comuns. A FCM para Q será 0. Quando P e Q são combinadas, se as variáveis de Q são as mesmas variáveis usadas por M_2 e M_3, a FCM para P + Q será 0, uma vez que o número de interseções não vazias excederá o número de interseções vazias. Isso implica que $\mu(P + Q) > \mu(Q)$, violando a *propriedade 4*. Logo FCM não satisfaz a *propriedade 4*.

Seja P e Q duas classes tais que $\mu(P) = \mu(Q) = n$, e deixe R ser uma outra classe com $\mu(R) = r$.

$$\mu(P + R) = n + r - \delta, \text{ similarmente}$$
$$\mu(Q + R) = n + r - \beta.$$

Dado que δ e β são funções independentes, elas não serão iguais , ou seja, $\mu(P + R)$ não é igual a $\mu(Q + R)$, satisfazendo a *propriedade 5*. Para quaisquer duas classes P e Q, $\mu(P + Q) = n_P + n_Q - \delta$. Ou seja,

$$\mu(P + Q) = \mu(P) + \mu(Q) - \delta \text{ o qual implica que}$$
$$\mu(P + Q) \leq \mu(P) + \mu(Q) \text{ para todo P e Q.}$$

Logo a *propriedade 6* não é satisfeita.

4.6. Conclusão.

Neste capítulo foram apresentados os aspectos mais importantes 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. Além disso, foi apresentado um conjunto de propriedades que servirão para a avaliação das métricas. A definição destas propriedades serve para avaliar de forma mais rigorosa o conjunto de métricas escolhido.

Cada métrica é uma dentre várias que podem ser definidas pelos princípios ontológicos de Bunge [B77]. Por exemplo a cardinalidade do conjunto de propriedades de um objeto, a qual inclui métodos e variáveis pode ser definida como uma métrica. As métrica escolhidas e detalhadas anteriormente foram influenciadas por três critérios adicionais: Habilidade de encontrar propriedades analíticas, simpatia por pessoas desenvolvedoras de software, e desta forma fácil percepção dos pontos de vista e por último fácil de coletar dados de forma automatizada.

A Figura 11 mostra uma tabela que mapeia as métricas para os passos do projeto de Orientação por Objetos de Booch.

Métrica	Identificação	Semântica	Relacionamento
MPC	X	X	
PAH	X		
NDF	X		
AEC		X	X
RDC			X
FCM		X	

Figura 11: Mapeamento das métricas para os passos do projeto de OO de Booch.

Observando a Figura 11 pode-se notar que existe uma vantagem em se ter um conjunto de métricas, pois através de um conjunto definido é possível fazer múltiplas medidas em cima de um mesmo projeto. Outro fator que beneficia esta funcionalidade é a relação existente entre as métricas, as quais estão sendo amparadas por um mesmo conjunto de critérios: classe, herança, passagem de mensagem, encapsulação.

5. Avaliação do Impacto da Programação Orientada por Objetos na Qualidade do Software.

5.1. Introdução.

Este capítulo mostra para cada uma das quatro linguagens apresentadas no Capítulo 3, o seu comportamento perante as métricas utilizadas. Mostra quais os pontos positivos e negativos de cada linguagem em relação a cada uma das métricas e a cada um dos aspetos avaliados.

5.2. Smalltalk.

Em Smalltalk não existe herança múltipla, logo uma classe herda propriedades apenas de uma classe pai, esta característica é um ponto favorável quando não é desejado ter classes muito complexas, pois o número de métodos herdados é bem menor do que a classe que herda de várias outras classes (herança múltipla). Com isso perceber-se que classes implementadas em Smalltalk tendem a ter um número menor de *Métodos Ponderados por Classe* (MPC).

Em relação *Profundidade da Árvore de Herança* (PAH) e *Número de Filhos* (NDF), mais uma vez, a herança é um aspecto fundamental. Linguagens que não possuem herança múltipla tendem a ter árvores de herança mais profunda, logo a extensibilidade é um fator que pode ser prejudicado.

A métrica *Número de Filhos* (NDF) mostra qual é o impacto de uma determinada classe seus descendentes. Em relação à herança, o número de filhos tende a aumentar quando existe herança múltipla, logo Smalltalk é uma linguagem que beneficia a produção de classes com poucos filhos.

Uma vez que os objetos são uma representação do mundo real, deve-se respeitar a hierarquia do mundo real no momento de implementar uma determinada classe, com isso a herança se torna algo natural e uma alteração em uma classe superior irá interferir de forma muito pequena nas classes herdeiras, facilitando assim a manutenção do software.

Uma das características mais expressivas de Smalltalk está relacionada com as métricas *Acoplamento Entre Classes* (AEC) e *Resposta de uma Classe* (RDC). Independente da aplicação ou contexto, em Smalltalk toda iteração entre entidades em tempo de execução é feita através de passagem de mensagem entre objetos, logo o acoplamento entre as classes é

alto, uma vez que um objeto passa uma mensagem a outro objeto temos como resposta uma ação.

Este comportamento em Smalltalk se deve ao fato de variáveis escalares simples, tais como: inteiros, reais, booleanos, caracteres, etc. serem objetos, assim como construções de controle de fluxo tais como *if, while* e *repeat* [S99]. Logo, invocações a estes objetos são feitos através de passagem de mensagem. Esta característica da linguagem faz com que aplicativos desenvolvidos em Smalltalk tornem mais lentos do que aplicativos construídos em outras linguagens, portanto para se obter um software mais rápido o projetista ou implementador da classe deve diminuir o acoplamento entre as classes de forma a promover a encapsulação e a modularidade, além de diminuir o número de resposta de uma classes, pois quanto maior o número de ações que uma determinada classe pode responder a um pedido feito por outra classe, maior será a complexidade da mesma [S99] .

Em relação a métrica de *Falta de coesão nos métodos* (FCM), Smalltalk não causa qualquer tipo de influencia devido a uma característica específica, logo o projetista ou implementador é o responsável por criar métodos que possuem coesão, facilitando o reuso.

Smalltalk apresenta recurso de classes abstratas, o qual possui as seguintes características: nenhum objeto instancia uma classe abstrata. Elas sempre são instancias de uma subclasse da classe abstrata. Métodos contidos em uma classe abstrata representam o protocolo comum para todos as subclasses. Subclasses podem, através do polimorfismo, redefinir métodos e adicionar novas propriedades. E por último, as classes abstratas provêm uma organizações hierárquica lógica que serve como base para classes relacionadas.

Como dito anteriormente, em comparação a linguagens imperativas compiladas convencionais, os programas Smalltalk equivalentes são significativamente mais lentos. Ainda que seja teoricamente interessante que a indexação de vetores e laços possam ser oferecidos dentro do modelo de passagem de mensagem, a eficiência é um fator importante na avaliação das linguagens de programação. Portanto, a eficiência será evidentemente uma questão a ser considerada na maioria das discussões sobre aplicabilidade prática da Smalltalk.

A vinculação dinâmica da Smalltalk impede que erros de tipo sejam detectados até a execução. Um programa pode ser escrito e compilado ainda que inclua mensagens a métodos não existentes. Isso provoca, posteriormente no desenvolvimento, um número de reparos de erros muito maior do que ocorreria em uma linguagem com tipos estáticos.

Smalltalk é uma linguagem pequena, mas com um grande sistema. A sintaxe da linguagem é simples e muito regular. Ela é um bom exemplo do poder que pode ser oferecido por uma linguagem pequena se for construída em torno de um conceito simples, mas poderoso [S99]. No caso da Smalltalk, esse conceito é que a programação pode ser feita usando somente uma hierarquia de classes construída utilizando-se herança, objetos e passagem de mensagens.

5.3. Eiffel.

Assim como Smalltalk, Eiffel é uma linguagem Orientada por Objeto pura pelo fato de ter sido projetada especificamente para suportar programação OO e não se baseia em nenhuma linguagem existente. Esta característica fez com que Eiffel fosse projetada para ser uma linguagem que proporciona um código de alta qualidade, confiável e principalmente reutilizável [S99] .

Eiffel é uma linguagem que permite herança múltipla, a qual deve ser usada com bastante cuidado, pois embora seja um recurso poderoso, a herança múltipla afeta a *Métodos Ponderados por Classe* (MPC), *Profundidade da Árvore de Herança* (PAH) e *Número de Filhos* (NDF) o que pode dificultar a extensibilidade, mas que usada de forma correta ajuda de maneira significativa a reusabilidade e a extensibilidade simultaneamente. Além disso a herança em Eiffel permite que se use alguns controles de visibilidade das propriedades afim de promover a encapsulação. Esses controles de visibilidade funcionam usando a palavra reservada *feature*, na qual pode-se anexar um qualificador, se o qualificador for *{none}* as propriedades que uma classe possui são ocultos tanto para a subclasses como para os clientes. Se o nome da classe for utilizado como qualificador, as propriedades são ocultas aos clientes, mas visíveis às subclasses, já se o qualificador for vazio, as propriedades são visíveis tanto para os clientes como para as subclasses, além disso é possível explicitar quais as classes que podem ver aquela propriedades. A herança múltipla juntamente com o recurso de *feature*, torna a herança em Eiffel ainda mais poderosa, pois é possível controlar no nível da herança as propriedades que serão vistas por outras classes, assim o número de métodos por classe pode diminuir significativamente (*Métodos Ponderados por Classe* (MPC)), deixando somente o que é necessário àquela classe. Além disso, este recurso faz com que o número de métodos redeclarados diminua, o que facilita a entendibilidade do código. Pelo fato da herança múltipla em Eiffel ser bem projetada faz com que a extensibilidade seja beneficiada.

Outro ponto beneficiado com o recurso de *feature* é o *Acoplamento Entre Classes* (AEC), pois o controle do acoplamento torna-se algo simples e fácil de ser feito.

Um outro recurso interessante de Eiffel é o *ANY*. Todas as classes declaradas em Eiffel são herdeiras da classe *ANY*. A vantagem deste recurso é que algumas propriedades desta classe podem ser redefinidas e como conseqüência todas as classes declaradas a partir de então herdaram as propriedades que foram redefinidas. Isto é importante pois dependendo das características das classes que estão sendo criadas é possível diminuir o número de elementos da árvore de herança, com isso uma classe na árvore de herança pode diminuir sua profundidade e com isso facilitar a extensibilidade, além do mais, o número de filhos também pode diminuir. Mesmo que o número de filhos de uma determinada classe seja grande, o que não é bom, pois dificulta a manutenção da classe, usando o recurso de *feature*, citado anteriormente, fica mais fácil de impedir que alterações sejam propagadas.

Quanto à resposta de uma classe e a coesão entre os métodos de uma classe, Eiffel não possui nenhuma característica que os prejudique ou os beneficie, ficando a cargo do projetista da classe ficar atento a estes pontos. Já o acoplamento das classes está diretamente ligado ao controle de visibilidade das propriedades da mesma através do recurso *feature*, como foi dito anteriormente. Esse recurso em Eiffel deve ser destacado, pois, embora todas as linguagens avaliadas possuam controle de visibilidade, o recurso de visibilidade em Eiffel é o mais elegante e de mais fácil entendimento, o que facilita e torna correto o seu uso.

Atualmente Eiffel apresenta uma biblioteca de classes que faz com que a linguagem se torne mais ampla. Suporte a softwares cliente-servidor, a banco de dados relacionais e manipulados por OO, suporte a análise léxica e *parsing*, à computações matemáticas orientadas por objetos, dentre outros. Essas bibliotecas são mais um dos recursos que beneficiam a reusabilidade em sistemas desenvolvidos em Eiffel [S99] .

5.4.C++.

C++ não é uma linguagem OO pura, e ainda está bem longe de ser. Uma das principais razões é o fato de C++ ser uma evolução do C, e como conseqüência, ele deve ser compatível com a versão anterior, ou seja, ser compatível com o C (backward compatible) [S99] . Isso o torna híbrido, suportando tanto programação baseada em procedimentos como programação OO.

Enquanto Smalltalk não permite que o usuário exerça qualquer controle de acesso sobre variáveis e sobre os métodos de instância em uma classe, o C++ fornece uma gama de controles de acesso [S99] . Alguns servem para controlar aquilo que o cliente pode e não pode ver na definição de classes. Outros servem para controlar o acesso por subclasses [S99].

C++ permite herança múltipla, uma classe pode ter uma classe pai ou ser uma classe órfã, sem uma super classe, o que não é permitido em Smalltalk nem em Eiffel.

Todas ou somente algumas propriedades podem ser herdadas da classe pai, além disso podem ser modificadas pela herdeira. A acessibilidade dos membros herdados pela subclasse pode ser diferente da acessibilidade dos membros correspondentes na classe pai [S99]. Deve-se ter bastante cuidado ao utilizar os modos de acesso a membros de uma classe, os quais podem ser público, privados ou protegidos. Em uma hierarquia de classes, a classe privada derivada interrompe o acesso a todos os membros de todas as classes ancestrais a todas as suas sucessoras, e os protegidos podem ou não ser acessíveis às subclasses subseqüentes [S99].

Note que o recurso de herança em C++ é bem confuso para uma linguagem OO, ela prejudica de muitas formas o reuso e a manutenção do software. Em relação à métrica de *Métodos Ponderados por Classe* (MPC), uma classe em C++ pode-se tornar muito complexa pelo fato de poder utilizar código C dentro do C++ além de C++ permitir herança múltipla. O *Número de Filhos* (NDF) e *Profundidade da Árvore de Herança* (PAH) são influenciados pelo fato de uma classe poder ou não ter uma super classe, o que causa uma hierarquia de classes irregular. Além disso em relação ao controle de acesso, a herança em C++ é bem mais confusa, o que dificulta a encapsulação, pois a *Resposta De uma Classe* (RDC) e *Acoplamento Entre Classes* (AEC) podem tornar-se confusos. C++ é uma linguagem que fornece ao programador acesso a muitos detalhes o que a torna extremamente mais complexa e mais susceptível a erros. C++ permite classes órfãs, herança simples, herança múltipla, um controle de acesso confuso e ainda um novo conceito que é o de classes amigas, as quais são necessárias quando é preciso escrever um subprograma que deve acessar os membros de duas classes diferentes.

C++ é muito grande e complexa, não foi desenvolvida especificamente para a OO e com isso não apresenta recursos poderosos para se obter os objetivos necessários de qualidade para software.

5.5. Java.

Assim como acontece em C++, Java não utiliza exclusivamente objetos, porém Java está mais próximo de uma linguagem OO do que C++, uma vez que somente tipos primitivos tais como: booleano, caractere e numérico não são objetos [S99] .

Assim como Smalltalk e Eiffel, todas as classes de Java devem ter uma pai, o que não ocorre em C++. No caso de Java todas as classes são subclasses da classe *Object* (classe raiz) [S99] .

Java não permite herança múltipla, porém inclui uma espécie de classe virtual chamada interface, a qual proporciona uma versão da herança múltipla, logo tendo as mesma influências na *Profundidade da Árvore de Herança* (PAH), *Número de Filhos* (NDF) e *Métodos Ponderados por Classe* (MPC). Um controle interessante em Java é o recurso *final*. Um método final não pode ser sobreposto em nenhuma classe descendente, assim como uma classe final não pode ser pai de nenhuma subclasse. Este recurso é interessante para se diminuir o número de filhos de uma determinada classe, assim como controlar o tamanho da árvore de herança. Também serve para controlar o número de método de uma determinada classe tornando-a menos complexa.

Um outro ponto positivo de Java é o controle de acesso, o qual é bem menos complexo do que o do C++. Além disso Java se destaca em relação ao encapsulamento, o qual possui o elegante conceito de pacotes. As classes podem ser organizadas em pacotes, o que torna o controle de visibilidade poderoso, com isso a *Acoplamento Entre Classes* (AEC) e *Resposta de uma Classe* (RDC) ficam mais fáceis de serem controlados e a encapsulação é garantida a fim de obter modularidade.

5.6. Influências do Polimorfismo nas Linguagens Avaliadas.

Esta seção mostra as influências do recurso de polimorfismo na qualidade do software e quais polimorfismos estão presentes nas quatro linguagens avaliadas: Smalltalk, Eiffel, C++ e Java.

O recurso de polimorfismo está ligado diretamente com a reusabilidade e com a extensibilidade do software. O polimorfismo nada mais é do que a capacidade de um mesmo operador, método, ou até mesmo uma classe, de exercer ações diferentes sobre objetos diferentes. Abstrações operam uniformemente em valores de tipos diferentes [B03]. Na OO os

tipos de polimorfismo mais importantes são o polimorfismo de inclusão que trata a herança e o polimorfismo de sobrecarga. A vinculação dinâmica por meio do polimorfismo é um recurso poderoso [S99]. Com esse tipo de recurso além de aumentar o grau de reusabilidade, a manutenção do software fica muito mais fácil, diminuindo a probabilidade de erros e tornando o software mais robusto e correto. As classes abstratas estão ligadas diretamente com o polimorfismo.

A seguir é apresentada uma visão geral do polimorfismo nas quatro linguagens avaliadas:

- **Smalltalk:** A verificação de tipos em Smalltalk é feita dinamicamente, ou seja, em tempo de execução, seu único objetivo é assegurar que a mensagem coincida com o método. As variáveis não são tipificadas, logo qualquer nome pode ser vinculado a qualquer objeto, como conseqüência imediata, Smalltalk suporta polimorfismo dinâmico. Os códigos em Smalltalk não se preocupam quanto ao tipo das variáveis, desde que elas sejam consistentes. Como ponto central desta explicação tem-se que, contanto que os objetos referenciados em uma expressão tenham métodos para as mensagens da expressão, os tipos dos objetos são irrelevantes, significando assim que todo o código é genérico, nenhum está ligado a um tipo particular [S99].
- **Eiffel:** O polimorfismo e a vinculação dinâmica em Eiffel é bem interessante, este destaque se dá devido a alguns fatores: a redefinição de propriedades é explícita, a linguagem é fortemente tipada, ou seja, o compilador pode verificar estaticamente se a aplicação de uma determinada propriedade está correta, isto significa que a linguagem usa tanto vinculação dinâmica quanto tipificação estática. A vinculação dinâmica garante que se mais de uma versão de uma rotina é aplicável, então a versão correta é selecionada. Já a tipificação estática garante que existe pelo menos uma versão correta. Logo, vale ressaltar que o polimorfismo em Eiffel se apresenta de uma forma muito elegante e eficiente, o que favorece ainda mais para a qualidade do software.
- **C++:** C++ apresenta vinculação estática e dinâmica. Suas funções são vinculadas estaticamente a uma definição de função, mas uma variável ponteiro ou referência com o tipo da classe básica pode ser usada para apontar para objetos de qualquer classe derivada daquela, dando suporte ao polimorfismo. Quando este tipo de variável é usada para chamar uma função que está definida em uma das classes derivadas, a vinculação deve ser feita dinamicamente à definição da função correta [S99]. Todas as

classes que forem dinamicamente vinculadas devem ter seu nome precedido da palavra reservada *virtual*, a qual deve ser a classe base. Toda classe que possua uma função virtual pura é uma classe abstrata. Nota-se que a polimorfismo de C++ é bem parecido com o de Eiffel, o que difere os dois é a sintaxe da linguagem, neste ponto quem ganha é Eiffel devido sua linguagem simples e de fácil leitura [S99].

- **Java:** O polimorfismo em Java é bem parecido com o de Smalltalk, uma vez que todas as vinculações de métodos são feitas dinamicamente [S99]. O polimorfismo pode ser aplicado a qualquer método que seja herdado de uma super classe. Assim como nas demais linguagens o que faz com que o polimorfismo funcione em Java é a vinculação dinâmica, isto é, o compilador não gera o código para chamar um método em tempo de compilação, deixando para ser feito em tempo de execução na hora em que o método é chamado. Todos os métodos em Java são virtuais, ou seja, seguindo a nomenclatura do C++, são método que possuem vinculação dinâmica. Caso não se deseje que isto ocorra é preciso colocar a palavra *final* para que seja feita vinculação estática.

5.7. Comparações.

As Figuras 12 e 13 mostram os pontos negativos (N) e positivo (P) das linguagens avaliadas nas seis métricas apresentadas na Seção 4.5.

	Smalltalk		**Eiffel**	
	P	**N**	**P**	**N**
MPC	• Herança Simples • Classes Abstratas		• Herança Múltipla + Controle de visibilidade: *feature* • Herança Simples	• Herança Múltipla
PAH	• Classes Abstratas	• Herança Simples	• Herança Múltipla • Controle de visibilidade: *feature* • Recurso: *ANY*	
NDF	• Herança Simples		• Controle de visibilidade: *feature* • Recurso: *ANY* • Herança Simples	• Herança Múltipla
AEC	• Herança Simples	• Baseada em Mensagens	• Controle de visibilidade: *feature* • Herança Simples	• Herança Múltipla
RDC	• Herança Simples	• Baseada em Mensagens	• Herança Simples	• Dependente do Implementador
FCM	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador

Fig. 12: Quadro comparativo I

	C++		Java	
	P	**N**	**P**	**N**
MPC	• Herança Simples	• Classe Orfã • Herança Múltipla • Modos de Acesso • Utilização de Código C • Classes Amigas	• Herança Simples • Recurso: *final*	• Interface (simulação de herança múltipla)
PAH	• Herança Múltipla • Classes Amigas	• Classe Orfã • Herança Simples • Modos de Acesso	• Interface (simulação de herança múltipla) • Recurso: *final*	• Herança Simples
NF	• Herança Simples	• Classe Orfã • Herança Múltipla • Modos de Acesso	• Herança Simples • Recurso: *final*	• Interface (simulação de herança múltipla)
AEC	• Herança Simples	• Herança Múltipla • Modo de Acesso • Classes Amigas	• Herança Simples • Recurso: *final* • Controle de Acesso Simples • Recurso: *Pacotes*	• Interface (simulação de herança múltipla)
RDC	• Herança Simples	• Modo de Acesso • Classes Amigas	• Herança Simples • Recurso: *final* • Controle de Acesso Simples • Recurso: *Pacotes*	
FCM	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador	• Dependente do Implementador

Fig. 13: Quadro comparativo II

A Figura 14 mostra outras comparações relevantes entre as linguagens. Vale observar que a linguagem Eiffel comparada no quadro abaixo é referente à ISE Eiffel, um aprimoramento da linguagem Eiffel original.

	Smalltalk	ISE Eiffel	C++	Java
Projeto por Contrato	Não possui	Projeto por Contrato	Não possui	Não possui
Verificação de Tipo	Tipada dinamicamente	Tipada estaticamente	Estaticamente tipada, embora a linguagem dê suporte à C-Style o qual permite violação das regras de tipo.	Na maioria das vezes tipada estaticamente, mas tipificação dinâmica é necessária para estruturas de "conteiner" genéricas.
Compilação	Historicamente baseada em interpretação, mas atualmente possui uma mistura de interpretação e compilação.	Combinação de compilação e interpretação no mesmo ambiente	Compilada	Mistura de interpretação e compilação "on-the-fly"
Eficiência do Código Gerado	Executáveis lentos, pois requer "Smalltalk Image"	Gera executáveis rápidos, mas não tão eficientes como C++	Gera executáveis rápidos	Possui muitos problemas de desempenho
Tratamento de Exceção	Possui	Possui	Possui	Possui
Herança	Herança simples	Herança múltipla	Herança múltipla, mas com vários problemas	Herança simples, embora a herança múltipla possa ser simulada através das "interfaces"
Facilidade de Uso	No geral possui uma sintaxe "bizarra"	Clara, simples e sintaxe legível	Sintaxe complexa	Sintaxe parecida com C++
Padronização de estilo de biblioteca	Ênfase na consistência	Bibliotecas com vocabulário padronizado	Ampla variação	?
Gerência de Memória	Possui coletor de lixo	Coletor de lixo/Gerência automática de memória	Não possui coletor de lixo	Possui coletor de lixo
Escopo no ciclo de vida do software	A maior parte está focada na implementação	Ambiente de desenvolvimento visual sem emenda	Somente implementação de endereçamento	Focado na implementação
Software matemático	Bibliotecas disponíveis	Bibliotecas disponíveis	Bibliotecas disponíveis	Bibliotecas disponíveis ou em construção
Interoperabilidade	Sem padronização de interface para a linguagem C	Linguagem e ambiente padronizados para interação com C e C++	Boa interoperabilidade com C	Métodos nativos
Grau de OO	Puramente OO	Puramente OO	Híbrida	Apresenta um alto grau de OO, mas não é puramente OO

Fig. 14: Quadro comparativo II.
Adaptado de: http://archive.eiffel.com/doc/manuals/technology/oo_comparison/page.html

5.8. Conclusão.

Neste capítulo foi apresentado as principais características de cada linguagem, bem como estas características influenciam na qualidade do software. As métricas escolhidas no Capítulo 4 aparecem aqui para dar um limite na comparação das linguagens, fazendo com que estas comparações fiquem todas no âmbito da OO. As características aqui determinadas, servem como base na escolha de determinada linguagem para o desenvolvimento de determinado sistema, de forma que a qualidade seja atingida.

6. Conclusão

Este trabalho teve como objetivo 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 analisada, desta forma possibilitando uma análise mais rigorosa e eficiente de quando utilizar uma determinada linguagem em detrimento de outra. Este trabalho serve como um guia e um conjunto de critérios para a utilização de uma linguagem OO em um determinado problema, visto que nem todas as linguagens Orientadas por Objetos possuem características suficientes para a produção de softwares de boa qualidade. Além disso, vale ressaltar, os *pontos de vista* dos desenvolvedores, assim como seus princípios de programação, os quais são de grande importância para o auxílio na produção de softwares de qualidade. Logo, bons desenvolvedores aliados à uma linguagem OO adequada podem produzir software extremamente eficientes e de excelente qualidade. Assim como uma linguagem que não se aplica ao contexto em que é utilizada faz com que um projeto saia caro e muito ineficiente.

Portanto o uso de uma linguagem OO na produção de um software deve ser muito bem estudada, pois não basta que uma linguagem seja do paradigma da Orientação por Objetos, ela dever oferecer recursos para alcançar o grau de qualidade proposto pela Engenharia de Software.

A Engenharia de Software prega a qualidade no desenvolvimento de novos sistemas, para isso ela estabelece uma 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 engenharia de software, 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 atingir nossos objetivos foi feito um estudo baseado em alguns recursos oferecidos pelas linguagens orientadas por objetos, tais como: classe, herança, encapsulação e passagem de mensagem e polimorfismo. Com esses recursos foi possível avaliar de uma maneira homogênea todas as linguagens, sem correr o risco de avaliar recursos que não possuem qualquer relação com a OO.

Após todas as comparações, ficou evidente que nem todas as linguagens orientadas por objetos são adequadas para se atingir um bom grau de qualidade no software desenvolvido. Como exemplo citamos C++, que embora seja uma linguagem que produza executáveis muito

eficientes, não possui, a partir de seus recursos, características que a tornem uma linguagem OO que produza softwares de qualidade. O amplo grau de liberdade de controle que a linguagem permite ao desenvolvedor a torna complexa e susceptível a erros. As vantagens mais expressivas do uso de C++ está na eficiência dos programas e na sua portabilidade, uma vez que a velocidade do C está presente [S99]. Logo, C++ é indicada quando o software em questão necessita de eficiência ao invés de qualidade. Sistemas desenvolvidos em C++, dificilmente atingirão uma alto grau de qualidade pelas definições de OO e da Engenharia de Software [S99].

Por outro lado, tem-se linguagens OO extremamente qualificadas no propósito que elas desejam atender. Eiffel é uma linguagem muito elegante e que apresenta recursos extremamente poderosos e de fácil entendimento, logo se torna uma linguagem rica em características que permitem a produção de software de boa qualidade. É uma linguagem que pode ser aplicada a qualquer tipo de problema, ela oferece bibliotecas que fazem com que a linguagem seja aplicável a qualquer escopo de trabalho, além do mais gera executáveis, não tão eficientes como C++, mas mais eficientes do que Smalltalk e Java, pois grande parte do código escrito em Eiffel é compilado. Além disso, diferente de C++, é uma forte candidata a ser usada por desenvolvedores em início de carreira, pois apresenta uma sintaxe muito simples e intuitiva, facilitando o entendimento e assim, diminuindo a probabilidade de erros no software, os quais podem muitas vezes serem detectados na fase de teste, o que torna caro o reparo. Além do mais Eiffel apresenta um tratamento de exceção muito eficaz e elegante, o que torna o software em Eiffel mais robusto e correto.

Smalltalk destina-se a produção de softwares que simulam o mundo real, tais como ferramentas de ensino, automação de escritórios, etc. Implementar softwares científicos ou que utilizem uma gama muito grande de operações aritméticas, Smalltalk não é uma boa indicação, pois devido a sua composição pura em objetos e por ser interpretada por uma "máquina virtual", ela se torna uma linguagem pouco eficiente. Possui muitos recursos para produção de software de qualidade, perdendo para Eiffel somente pela sintaxe que foge dos padrões de linguagens de programação mais comuns, e, pela pouca eficiência do código gerado. Embora estes dois últimos critérios não tenham sido alvo do estudo, eles aparecem como critérios de desempate entre Smalltalk e Eiffel.

Por último temos a linguagem Java, que aparece como sendo uma linguagem de fácil entendimento, com recursos interessantes para modularidade e que segue de maneira

significativa os princípios da OO. Java pode ser considerada como sendo a evolução de C++. Java possui muitas características baseadas em C++, mas com um tratamento muito mais orientado a objetos do que esta última, além de possuir uma sintaxe mais simples, permitindo a escrita de códigos mais limpos.

Java produz programas menos eficientes do que programas desenvolvidos em outras linguagens, devido a passagem de mensagens entre os objetos e pelo fato de Java executar programas em uma máquina virtual. Porém Java possui a vantagem e ser compatível com várias plataformas por ser compilada em bytecode [S99]. A maior aplicabilidade de Java são as aplicações WEB, devido sua portabilidade.

Apêndice A

Benchmark

A.1. Introdução.

Este apêndice apresenta o algoritmo para resolução do problema das 8 rainhas em um tabuleiro de xadrez. Os algoritmos apresentados estão em Smalltalk, Eiffel, C++ e Java. O objetivo deste benchmark é mostrar as características referentes à qualidade e legibilidade do software. Logo, características como eficiência do executável gerado, do compilador ou outras características fogem do foco de estudo deste trabalho e não serão abordadas aqui.

A.2. Benchmarks nas Diversas Linguagens Avaliadas.

A seguir será mostrado o algoritmo que resolve o problema das 8 rainhas em um tabuleiro de xadrez e como ele se apresenta nas quatro linguagens.

- **Smalltalk:**

```
setColumn: aNumber neighbor: aQueen
                    " initialize the data fields "
          column := aNumber.
          neighbor := aQueen.
          row := 1.

canAttack: testRow column: testColumn | columnDifference |
     columnDifference := testColumn - column.
     (((row = testRow) or:
          [ row + columnDifference = testRow]) or:
          [ row - columnDifference = testRow])
               ifTrue: [ " true ].
      ↑neighbor canAttack: testRow column: testColumn

advance
          " first try next row "
       (row < 8)
          ifTrue: [ row := row + 1. ↑ self findSolution ].
          " cannot go further, move neighbor "
       (neighbor advance) ifFalse: [ ↑ false ].
          " begin again in row 1 "
        row := 1.
        ↑ self findSolution


findSolution
      [ neighbor canAttack: row column: column ]
          whileTrue: [ self advance ifFalse: [ ↑ false ] ].
      ↑ true

result
          ↑ neighbor result; addLast: row
```

```
solvePuzzle | lastQueen |
        lastQueen := SentinelQueen new.
        1 to: 8 do: [:i | lastQueen := (Queen new)
            setColumn: i neighbor: lastQueen.
            lastQueen findSolution ].
        ↑ lastQueen result
```

Fonte: ftp://ftp.cs.orst.edu/pub/budd/oopintro/2ndEdition/queens/

- **Eiffel:**

```
class CHESS_BOARD
        inherit
                D2_ARRAY[CHESS_PIECE]
                rename
                        valid_coordinates as is_on_board,
                        item_at as piece_at,
                        put_at as attach
                end
        creation
                create
                        -- Must be initialised as 8 * 8 matrix
        feature
                detach(row:INTEGER; col:INTEGER);
                        -- removes piece from board
                        require
                                is_on_board(row,col)
                                ensure
                                        not is_occupied(row,col)
                is_occupied(row,col:INTEGER):BOOLEAN
                        -- true if a piece is attached at position specified
                        require
                                is_on_board(row,col)
                can_be_reached(row,col:INTEGER;colour:INTEGER):BOOLEAN
                        -- checks if a piece can be reached by
                        -- another of the opposite colour
                        require
                                is_on_board(row,col)
                is_clear_path(row,col,dest_row,dest_col:INTEGER):BOOLEAN
                        -- checks path between two squares on board
                        require
                                is_on_board(row,col);
                                is_on_board(dest_row,dest_col);

                invariant
                        rows = 8;
                        columns = 8
 end -- CHESS_BOARD

class CHESS_VIEW
                feature
                display(board:CHESS_BOARD) is
                        do
                        from c := 0
                        until c = 8
                                loop
                                c := c +1;
                                io.put_string("    "); -- 5 spaces
```

```
                    io.put_integer(c);
                    end -- loop
            io.put_new_line
            io.put_string ("_____")
            -- 48 underscore characters
            io.put_new_line;
            from r := 0;
            until r = 8
            loop
                    io.put_string(" |   |   |   |   |   |   |   |");
                    -- 5 spaces between each vertical bar
                    io.put_new_line;
                    r := r +1;
                    io.put_integer(r);
                    io.put_character('|');
                    from c := 0
                    until c = 8
                    loop
                            c := c+1
                            if board.is_occupied(r,c)
                                    then board.piece_at(r,c).display
                                    else io.put_string("    ")
                            end -- if
                    io.put_string("  |");
                    end -- loop
                    io.put_new_line;
                    io.put_string(" |___|___|___|___|___|___|___|___|");
                    -- vertical bars separated by 5 underscore
                    -- characters
                    io.put_new_line;
    end -- display
end -- CHESS_VIEW
```

Fonte: ftp://ftp.cs.orst.edu/pub/budd/oopintro/2ndEdition/queens/

- **C++:**

```
# include <iostream>
# define bool int

class queen {
public:
        // constructor
    queen (int, queen *);

        // find and print solutions
    bool findSolution();
    bool advance();
    void print();
```

```
private:
        // data fields
    int row;
    const int column;
    queen * neighbor;

        // internal method
    bool canAttack (int, int);
};

queen::queen(int col, queen * ngh)
    : column(col), neighbor(ngh)
{
    row = 1;
}

bool queen::canAttack (int testRow, int testColumn)
{
        // test rows
    if (row == testRow)
        return true;

        // test diagonals
    int columnDifference = testColumn - column;
    if ((row + columnDifference == testRow) ||
        (row - columnDifference == testRow))
            return true;

        // try neighbor
    return neighbor && neighbor->canAttack(testRow, testColumn);
}

bool queen::findSolution()
{
        // test position against neighbors
    while (neighbor && neighbor->canAttack (row, column))
        if (! advance())
            return false;

            // found a solution
    return true;
}

bool queen::advance()
{
    if (row < 8) {
        row++;
        return findSolution();
        }

    if (neighbor && ! neighbor->advance())
        return false;

    row = 1;
    return findSolution();
}

void queen::print()
{
    if (neighbor)
        neighbor->print();
    cout << "column " << column << " row " << row << '\n';
}
```

```
void main() {
        queen * lastQueen = 0;

        for (int i = 1; i <= 8; i++) {
                lastQueen = new queen(i, lastQueen);
                if (! lastQueen->findSolution())
                        cout << "no solution\n";
                }

        lastQueen->print();
}
```

Fonte: ftp://ftp.cs.orst.edu/pub/budd/oopintro/2ndEdition/queens/

- **Java:**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Queen {
                // data fields
        private int row;
        private int column;
        private Queen neighbor;

                // constructor
        Queen (int c, Queen n) {
                        // initialize data fields
                row = 1;
                column = c;
                neighbor = n;
                }

        public boolean findSolution() {
                while (neighbor != null && neighbor.canAttach(row, column))
                        if (! advance())
                                return false;
                return true;
                }

        public boolean advance() {
                if (row < 8) {
                        row++;
                        return findSolution();
                        }
                if (neighbor != null) {
                        if (! neighbor.advance())
                                return false;
                        if (! neighbor.findSolution())
                                return false;
                        }
                else
                        return false;
                row = 1;
                return findSolution();

                }

        private boolean canAttach(int testRow, int testColumn) {
                int columnDifference = testColumn - column;
```

```
                if ((row == testRow) ||
                        (row + columnDifference == testRow) ||
                        (row - columnDifference == testRow))
                                return true;
                if (neighbor != null)
                        return neighbor.canAttach(testRow, testColumn);
                return false;
                }

        public void paint (Graphics g) {
                        // first draw neighbor
                if (neighbor != null)
                        neighbor.paint(g);
                        // then draw ourself
                        // x, y is upper left corner
                int x = (row - 1) * 50 + 10;
                int y = (column - 1) * 50 + 40;
                g.drawLine(x+5, y+45, x+45, y+45);
                g.drawLine(x+5, y+45, x+5, y+5);
                g.drawLine(x+45, y+45, x+45, y+5);
                g.drawLine(x+5, y+35, x+45, y+35);
                g.drawLine(x+5, y+5, x+15, y+20);
                g.drawLine(x+15, y+20, x+25, y+5);
                g.drawLine(x+25, y+5, x+35, y+20);
                g.drawLine(x+35, y+20, x+45, y+5);
                g.drawOval(x+20, y+20, 10, 10);
                }

        public void foo(Queen arg, Graphics g) {
                if (arg.row == 3)
                        g.setColor(Color.red);
                }
}

public class QueenSolver extends JFrame {

        public static void main(String [ ] args) {
                QueenSolver world = new QueenSolver();
                world.show();
        }

        private Queen lastQueen = null;

        public QueenSolver() {
                setTitle("8 queens");
                setSize(600, 500);
                for (int i = 1; i <= 8; i++) {
                        lastQueen = new Queen(i, lastQueen);
                        lastQueen.findSolution();
                        }
                addMouseListener(new MouseKeeper());
                addWindowListener(new CloseQuit());
                }

        public void paint(Graphics g) {
                super.paint(g);
                        // draw board
                for (int i = 0; i <= 8; i++) {
                        g.drawLine(50 * i + 10, 40, 50*i + 10, 440);
                        g.drawLine(10, 50 * i + 40, 410, 50*i + 40);
                }
                g.drawString("Click Mouse for Next Solution", 20, 470);
                        // draw queens
```

```
                lastQueen.paint(g);
                }

        private class CloseQuit extends WindowAdapter {
                public void windowClosing (WindowEvent e) {
                        System.exit(0);
                }
        }

        private class MouseKeeper extends MouseAdapter {
                public void mousePressed (MouseEvent e) {
                        lastQueen.advance();
                        repaint();
                }
        }
}
```

Fonte: ftp://ftp.cs.orst.edu/pub/budd/oopintro/2ndEdition/queens/

Como pode ser observado, o código menor e mais claro é aquele mostrado em Smalltalk, porém é um código que foge dos padrões de programação existentes, as formas com que são declarados laços de repetição, comando condicionais e atribuições é bem diferente. Já Eiffel apresenta um código também um pouco diferente, mas com uma forma de distribuição e sintaxe mais legíveis. Já Java e C++ embora bem parecidos, fica claro que Java é mais simples e mais objetiva do que C++. Nota-se que os controles de visibilidade em Java são mais fáceis e mais usados do que em C++.

Código menor não significa código bom, embora a legibilidade e a entendibilidade sejam fatores de qualidade importantes, um código pequeno não significa que seja fácil de se entender e como conseqüência de dar manutenção.

A.3. Conclusão.

Esses benchmarks estão exemplificando como são as formas de declaração e construção de classes e objetos nas várias linguagens. Servem como um ponto de referência para a observação dos códigos gerados nas diversas linguagens. Fica evidente que para se mostrar todos os fatores avaliados: classe, herança, passagem de mensagem e polimorfismo, de maneira significativa, seria preciso um sistema grande e com um número de classes e objetos significativos, além do mais teria que estar implementado nas quatro linguagens avaliadas, logo ficaria inviável de ser apresentado no contexto deste trabalho. Portanto os benchmarks aqui apresentados, como dito anteriormente, servem com um ponto inicial de observação das diferentes características das quatro linguagens.

7. Bibliografia.

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

[B73] Birtwistle, G. M.; Dahl, O-J; Myhrhaug, B.; Nygaard, K., *"Simula Begin"*. Philadelphia; Auerbach. 1973.

[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.

[B79] V. Basili e R. Reiter, "*Evaluating automable measures of software models*". IEEE Workshop Quantitative Software Models, Kiamesha, NY, 1979, pp. 107-116.

[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] MEYER, Bertrand. *Eiffel The Language.* Prentice Hall, New York, 1992.

[B92] 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.

[C01] HORSTMANN, Cay S.; CORNEL, Gary. *Core Java 2 – Volume I Fundamentos*, Markron Books. São Paulo, 2001.

[C91] J. C. Cherniavsky e C. H. Smith, "*On Weyuker's axioms for software complexity measures* ", IEEE Transactions on Software Eng, vol. 17, pp. 636-638, 1991.

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

[EF] *"Eiffel Software"*. Disponível em: http://www.eiffel.com

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

[FC] *"Free Online C and C++ Documentation Free online C/C++ guides, books, documentation, FAQ, tutorials, references, ebooks..."*.
Disponível em: http://www.thefreecountry.com/documentation/onlinecpp.shtml

[G85] Goldberg, Adele; Robson, David. *Smalltalk-80 The Langueg and Its Implementation*. Addison-Wesley Publishing Company, 1985.

[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>.

[L88] Pinson, Lewis J.; Wiener, Richard S.. *Na Introduction to Object-Oriented Programming and Smmaltalk.* Addison-Wesley Publishing Company, 1988.

[MM] Maia Luiz, Paulo; Morelli, Eduardo, "*Programação Orientada a Objetos*". Disponível em: < http://www.training.com.br/pub_prog_oo.htm>.

[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

[SM] "*www.smalltalk.org™*". Disponível em: http://www.smalltalk.org

[SMS] "*Sun Microsystems Brasil*". Disponível em: http://br.sun.com/

[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.

Belo Horizonte, 30 de Janeiro de 2004

_____

Alexander Thiago de Assis Oliveira Carvalho

_____

Mariza Andrade da Silva Bigonha