UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ULISSES BRISOLARA CORRÊA

Aplicação de Métricas de Software na Predição de Características Físicas de Software Embarcado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Luís Lamb
Orientador
Prof. Dr. Luigi Carro
Coorientador

Porto Alegre, Janeiro de 2011.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Corrêa, Ulisses Brisolara

Aplicação de Métricas de Software na Predição de Características Físicas de Software Embarcado / Ulisses Brisolara Corrêa – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

93p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2010. Orientador: Luis da Cunha Lamb.

1. Software para Sistemas Embarcados. 2. Sistemas Embarcados 3. Métricas Físicas 4. Métricas de Software I. Lamb, Luis II. Carro, Luigi. III. Título.

AGRADECIMENTOS

Agradeço a minha família que sempre apoiou minhas escolhas e entendeu minha ausência, em especial agradeço aos meus pais, Luiz Carlos e Vera Maria, por todo o empenho para que nós estudássemos e buscássemos por ambições maiores.

Agradeço ao CNPq pelo suporte financeiro a este trabalho, e ao apoio financeiro dos meus pais quando o apoio oficial faltou.

Agradeço a minha namorada, Larissa, por ter sido uma grande amiga e companheira durante esse processo. Em especial agradeço pela paciência dela nos últimos meses de escrita, quando o estresse estava realmente falando mais alto.

Agradeço aos colegas de laboratório pelas conversas, discussões e tudo mais. Em especial agradeço a Lisane e ao Július que sempre toparam discutir minhas ideias malucas e, ainda, abraçavam as maluquices.

Agradeço a meus orientadores pelo tempo dispensado à revisão deste trabalho.

Por último, mas não menos importante, agradeço a Deus que se manteve sempre por perto, mesmo quando minha fé não estava.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
ISO	International Organization for Standardization
JVM	Java Virtual Machine
MDA	Model Driven Architecture
MDD	Model Driven Development
MLP	Multi Layer Perceptron
MP3	MPEG Audio Layer 3
MPEG	Moving Picture Experts Group
OO	Orientação a Objetos
PC	Personal Computer
POO	Paradigma de Orientação a Objetos
RNA	Rede Neural Artificial
SoC	System on a Chip
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A complexidade dos dispositivos embarcados propõe novos desafios para o desenvolvimento de software embarcado, além das tradicionais restrições físicas. Então, a avaliação da qualidade do software embarcado e seu impacto nessas propriedades tradicionais torna-se mais importante. Conceitos como reúso abstração, coesão, acoplamento, entre outros atributos de software têm sido usados como métricas de qualidade no domínio da engenharia de software. No entanto, elas não têm sido usadas no domínio do software embarcado. No desenvolvimento de sistemas embarcados outro conjunto de ferramentas é usado para estimar as propriedades físicas, tais como: consumo de energia, ocupação de memória e desempenho. Essas ferramentas geralmente envolvem custosos processos de síntese e simulação. Nos complexos dispositivos embarcados atuais deve-se confiar em ferramentas que possam ajudar na exploração do espaço de projeto ainda nos níveis mais altos de abstração, identificando a solução que representa a melhor estratégia de projeto em termos da qualidade de software, enquanto, simultaneamente, atenda aos requisitos físicos. Neste trabalho é apresentada uma análise da correlação entre métricas de qualidade de software, que podem ser extraídas antes do sistema ser sintetizado, e as métricas físicas do software embarcado. Usando uma rede neural nós investigamos o uso dessas correlações para predizer o impacto que uma determinada modificação no software trará às métricas físicas do mesmo software. Esta estimativa pode ser usada para guiar decisões em direção a melhoria das propriedades físicas dos sistemas embarcados, além de manter um equilíbrio em relação às métricas de software.

Palavras-Chave: sistemas embarcados, engenharia de software para sistemas embarcados, métricas de qualidade de software, predição de características físicas.

Application of Software Quality Metrics to Predict Physical Characteristics of Embedded Systems

ABSTRACT

The complexity of embedded devices poses new challenges to embedded software development in addition to the traditional physical requirements. Therefore, the evaluation of the quality of embedded software and its impact on these traditional properties becomes increasingly relevant. Concepts such as reuse, abstraction, cohesion, coupling, and other software attributes have been used as quality metrics in the software engineering domain. However, they have not been used in the embedded software domain. In embedded systems development, another set of tools is used to estimate physical properties such as power consumption, memory footprint, and performance. These tools usually require costly synthesis-and-simulation design cycles. In current complex embedded devices, one must rely on tools that can help design space exploration at the highest possible level, identifying a solution that represents the best design strategy in terms of software quality, while simultaneously meeting physical requirements. We present an analysis of the cross-correlation between software quality metrics, which can be extracted before the final system is synthesized, and physical metrics for embedded software. Using a neural network, we investigate the use of these cross-correlations to predict the impact that a given modification on the software solution will have on embedded software physical metrics. This estimation can be used to guide design decisions towards improving physical properties of embedded systems, while maintaining an adequate trade-off regarding software quality.

Keywords: embedded systems, embedded software engineering, software quality metrics, neural networks, power consumption prediction, performance prediction.

1 INTRODUÇÃO

1.1 Motivação

Segundo Peter Marwedel, a era pós-PC (do inglês *Personal Computer*) está chegando, onde processadores e software estão sendo usados, cada vez mais, em dispositivos cada vez menores. Essa tendência emergente, de redução do tamanho físico dos dispositivos, tem sido referenciada pelo termo desaparecimento do computador (MARWEDEL, 2003).

Recentes lançamentos da indústria corroboram essa mudança gradual de caráter dos sistemas computacionais de grandes (com considerável capacidade computacional) para sistemas pequenos (com capacidade computacional limitada). Empresas de grande expressão no mercado de semicondutores como a Intel©[1] e a Via©[2] lançaram linhas de produtos voltados a sistemas de baixa capacidade computacional e baixo consumo energético. A primeira empresa citada apresentou recentemente o projeto de uma plataforma para o mercado de sistemas embarcados baseada em um sistema *multichip* em um único emcapsulamento que associa um processador Atom™ com uma unidade reconfigurável, um FPGA (do inglês *Field Programmable Gate Array*) da fabricante Altera (INTEL, 2010).

Porém, do ponto de vista do processo de desenvolvimento de software ainda não há uma convergência de interesse para esse novo modelo de sistemas computacionais. Embora, seja importante ressaltar que o processo de desenvolvimento de software venha sendo amplamente aprimorado ao longo dos anos (SOMMERVILLE, 2006), porém o foco desse avanço tem sido tornar possível que os sistemas desenvolvidos sejam cada vez mais confiáveis, seguros, manuteníveis, etc. Esses avanços, portanto, não abrangem técnicas que visem a otimização daqueles fatores de qualidade que possuem relação mais forte com as plataformas onde essas aplicações serão executadas, em outras palavras, esses avanços não contemplam otimização das aplicações quanto ao seu consumo de recursos físicas.

No entanto, fatores de qualidade relacionados ao desempenho do software terão cada vez mais importância nessa nova era e, no entanto, têm sido relegados atualmente nos processos de desenvolvimento de software. Ciclos de CPU (Unidade central de processamento - do inglês *Central Processing Unit*), energia e memória consumidos são

[1] http://edc.intel.com/

[2] http://www.via.com.tw/en/products/embedded/

exemplos de fatores que não têm sido levados em consideração, mas despontarão como restrições em projetos neste novo contexto (MARWEDEL, 2003). Isso se dá, pois, a maioria dos desenvolvedores de software se acostumou com o avanço computacional compatível com as predições da Lei de Moore (MOORE, 1965), e, assim sendo, o esforço de gerar o equilíbrio entre a qualidade do produto de software e a eficiência de consumo dos recursos computacionais da plataforma alvo é encarado com um esforço desnecessário.

Enquanto isso, nos sistemas embarcados o enfoque do processo de desenvolvimento tem um apego muito maior a esses fatores de qualidade preteridos nos processos de desenvolvimento de software convencional. Como consequência, no contexto de sistemas embarcados, a busca pelos fatores de qualidade do produto de software tende a ser relaxada em prol de sistemas que atendam às restrições de projeto (tempo de resposta, consumo de memória, etc.). Plataformas com recursos limitados, bem como sistemas computacionais com restrições rígidas de tempo de resposta, frequentemente conduzem o processo de desenvolvimento de software embarcado a uma abordagem dirigida a restrições de projeto, diferentemente do processo de desenvolvimento convencional, onde o foco são as boas práticas adotadas na engenharia de software (GANSSLE, 2008).

Dada à referida tendência de conversão de caráter dos sistemas computacionais, é válido salientar que os processos de desenvolvimento de software aplicados até então terão de convergir para um novo tipo de processo, onde a qualidade do produto de software não poderá mais ser vista apenas do ponto de vista da engenharia de software convencional, visto que os dispositivos da era pós-PC se assemelharão aos sistemas embarcados atuais. Nesse contexto, as restrições físicas também deverão ser consideradas no processo de desenvolvimento do software. Baseado nisso, desenvolver meios que facilitem o equilíbrio entre essas duas abordagens tem ganhado extrema importância.

Tendo em vista que as métricas de software são, geralmente, utilizadas para aferir a qualidade do produto de software, então, uma definição para o termo qualidade faz-se necessária para um melhor embasamento. A seguir é apresentada a definição extraída da Norma NBR 8402 da ABNT/ISO (do inglês *International Organization for Standardization*) (ABNT, 1994): “Qualidade é a totalidade das características de um produto ou serviço que se refletem na sua capacidade de satisfazer necessidades especificadas ou implícitas”.

Em geral, o conceito de qualidade por si só deixa espaço para ambiguidades, por isso frequentemente as disciplinas de engenharia buscam por representações numéricas dos fatores que possam indicar a qualidade das entidades analisadas (MÖLLER e PAULISH, 1992). No caso específico da engenharia de software, as métricas mais comuns para avaliar os fatores de qualidade de um produto de software serão discutidas em um capítulo posterior.

Como mencionado anteriormente, o processo de desenvolvimento de software tem sido aperfeiçoado constantemente. Esse aperfeiçoamento tem se dado através do desenvolvimento e aplicação de novas técnicas de engenharia, de tal forma que novos métodos têm sido propostos para todas as etapas de projeto, desde a análise de requisitos até a validação e teste do software. Uma dessas ferramentas, que tem sido amplamente explorada, é o emprego do uso de abstrações. Tanto o uso do paradigma de Programação Orientada a Objetos (POO), que é hoje o paradigma adotado na maioria

dos projetos de software (SOMMERVILLE, 2006), quanto à crescente adoção de métodos de projeto orientado a modelos (SELIC, 2006) exemplificam o emprego desse mecanismo.

A revelia dos esforços desenvolvidos para melhorar o processo de desenvolvimento de software para computadores normais, os projetos de sistemas embarcados, em geral, são desenvolvidos em níveis de abstração muito baixos. Isso é feito de forma que determinadas restrições de projeto possam ser alcançadas. Nesse contexto pode-se dizer que o desenvolvimento de software embarcado é dirigido a restrições de projeto, ou seja, este tipo de projeto do software é completamente desenvolvido para extrair o máximo desempenho no uso dos recursos da plataforma alvo, visto que esses sistemas não devem ter recursos que fiquem ociosos (MARWEDEL, 2003).

A avaliação quantitativa é um fator chave de qualquer processo de engenharia, pois permite medir e avaliar as propriedades de um produto; assim diferentes métricas para avaliar a qualidade de produtos de software têm sido propostas (AGGARWAL, SINGH, *et al.*, 2006). Estas métricas permitem avaliar conceitos como: reúso, abstração, coesão, acoplamento e outros atributos de qualidade de software.

No entanto, o desenvolvimento de software embarcado difere do software tradicional quanto às exigências de eficiência impostas ao projeto. Portanto, diferentemente do domínio de software convencional, as principais métricas adotadas para avaliar soluções de software embarcado são métricas físicas tais como desempenho, tamanho da memória, consumo de potência e peso. Além destas restrições, o projetista de software embarcado também tem de lidar com janelas de mercado cada vez mais estreitas e preocupar-se em produzir software de boa qualidade, em pouco tempo e com menor custo, de forma a tornar-se competitivo. Desta forma, características como reúso, manutenibilidade, tempo de projeto e custo também são importantes. Apesar de muitas metodologias suportarem a extração de métricas físicas através de ferramentas de simulação ou estimação, fatores como reúso, manutenibilidade e tempo de projeto não costumam ser avaliados no domínio de sistemas embarcados. Isto ocorre porque a cultura de desenvolvimento de sistemas embarcados se estabeleceu anos atrás, quando esses dispositivos tinham funções bem definidas, diferentemente dos dias atuais, onde esses dispositivos têm aplicações cada vez mais heterogêneas.

A busca pelo aumento da produtividade e da qualidade do software embarcado aumenta o interesse pela aplicação de métodos da engenharia de software no domínio de sistemas embarcados. Porém, devido às típicas restrições rígidas encontradas na maior parte dos projetos de sistemas embarcados, a comunidade de embarcados fica impedida de se beneficiar do uso de metodologias avançadas de engenharia de software. Por consequência, o processo de desenvolvimento de software é atualmente considerado o gargalo no projeto de sistemas embarcados, contabilizando 80% dos custos do desenvolvimento destes sistemas (JERRAYA, YOO, et al., 2003). Apesar do crescimento na aplicação de metodologias de engenharia de software, a prática atual do desenvolvimento de software embarcado é ainda insatisfatória, em particular na indústria (SRINIVASAN, DOBRIN e LUNDQVIST, 2009), (JERRAYA, YOO, et al., 2003) e (GRAAF, LORMANS e TOETENEL, 2003).

Métricas tradicionais de qualidade de software têm sido aplicadas com sucesso no processo de melhoraria da qualidade do software nos sistemas de informação convencionais, apontando para progressos no reúso e tempo de projeto.

Tradicionalmente, estas métricas ajudam projetistas a melhorar fatores de qualidade do produto de software, como abstração, reúso e manutenibilidade. Porém, algumas práticas consideradas como indicadas no desenvolvimento de software convencional podem causar impactos negativos nas métricas físicas.

Isto posto, em nossos experimentos buscamos por relações entre essas métricas, capazes de indicar a qualidade de uma determinada implementação, e as características físicas dessas implementações. Essas características, ou métricas de baixo nível (MANGALAMPALLI e JAIN, 2007), são os consumos efetivos de memória, energia e ciclos de CPU referentes a uma implementação em questão. Através do estudo dessas relações, pretende-se relacionar quais métricas de qualidade podem ser usadas em conjunto com as métricas físicas. Além disso, experimentos demonstram que as boas práticas de projeto orientado a objetos, apontadas pela engenharia de software, podem causar um impacto negativo nas propriedades físicas de um sistema embarcado, sendo necessário muitas vezes sacrificar atributos como reúso ou manutenibilidade a fim de obter um software mais eficiente em tempo de execução (MARWEDEL, 2003).

Segundo Smaalders (2006) e Sommerville (2006), as boas práticas da engenharia de software sugerem que não se devem gastar esforços em otimização em estágios iniciais do projeto e sim em um estágio final, assim mantendo o código mais limpo e inteligível, além de facilitar a portabilidade e, por consequência, o reúso do código. Portanto, uma abordagem que facilite essa etapa de otimização, automaticamente ou semi-automaticamente, é importante para reduzir o tempo de lançamento de um produto, ajudando, assim, o produto a se posicionar no mercado.

1.2 Objetivo Geral

Este trabalho foca na aplicação de métodos automáticos para a análise da relação entre métricas clássicas de qualidade de software em projetos orientados a objetos e métricas físicas, tradicionalmente adotadas pela comunidade de sistemas embarcados. O objetivo geral é encontrar a correlação entre essas métricas para que possam ser usadas pelo projetista de software embarcado na exploração do espaço de projeto e na busca de soluções de qualidade tanto em reúso e manutenibilidade quanto no que se refere à eficiência no uso de recursos. Desta forma, as métricas podem ser usadas para guiar os projetistas na busca pela solução que melhor atende os requisitos da aplicação e que possui melhor qualidade.

1.3 Objetivo Específico

Esse trabalho visa explorar novas formas de ajudar os desenvolvedores de software a realizarem esse equilíbrio entre a qualidade do produto de software e o seu consumo de recursos físicos na plataforma de hardware.

No âmbito da busca por esse equilíbrio entre as boas práticas da engenharia de software e as restrições às características físicas exigidas pelo projeto de sistemas embarcados o objetivo deste trabalho é apresentar uma forma de predizer os consumos físicos de aplicações, baseado nas métricas clássicas de qualidade do produto de software.

1.4 Organização do Texto

O restante deste trabalho está organizado como segue. O Capítulo 2 apresenta uma breve discussão sobre trabalhos correlatos. Depois, uma revisão sobre métricas de

qualidade de software, detalhando as métricas que serão usadas neste trabalho é apresentada no Capítulo 3. Então, os experimentos iniciais são apresentados no Capítulo 4. Em seguida, no Capítulo 5, são apresentados experimentos realizados com uma abordagem baseada em baixa variabilidade de dados de entrada das redes neurais. Após, no Capítulo 6, uma análise dos resultados obtidos pelos experimentos é apresentada. Por fim, o Capítulo 7 apresenta as conclusões e alguns dos possíveis trabalhos futuros.

2 TRABALHOS CORRELATOS

2.1 Introdução

A obtenção de métricas físicas de aplicações em curtos espaços de tempo tem sido uma área de pesquisa de grande atividade nos últimos anos, visto que os mercados de software tornaram-se cada vez mais competitivos, sobretudo no contexto do software embarcado, que, como discutido anteriormente, tem emergido como uma área de destaque no contexto dos sistemas computacionais modernos.

Ainda que poucos trabalhos abordem a avaliação da qualidade do software embarcado, relacionando métricas de qualidade e métricas físicas, buscou-se por trabalhos que complementem, ou se assemelhem parcialmente, com nossos objetivos. Nas seções a seguir discutiremos alguns trabalhos recentes e suas propostas no contexto deste trabalho.

2.2 DESEJOS (do inglês *DEsign of Software for Embedded Java with Object Support*)

Mattos (MATTOS e CARRO, 2007) propôs uma ferramenta para obtenção de estimativas de características físicas da execução de aplicações Java para diferentes versões da arquitetura FemtoJava (ITO, CARRO e JACOBI, 2001). Esta ferramenta tem como objetivo acelerar o processo de obtenção de métricas de baixo nível, como ciclos e energia consumidos pela execução de uma aplicação.

Diferentemente de outras ferramentas que oferecem maior precisão através da aplicação de modelos mais complexos dessas arquiteturas, como CACO-PS (BECK e CARRO, 2003), Mattos apresentou uma abordagem baseada na caracterização da execução de instruções Java no processador FemtoJava, nas versões multiciclo e *pipeline*, em termos de seus consumos individuais. A estimativa da ferramenta para o custo total de uma aplicação se dá, então, pelo acumulo dos custos individuais (em ciclos ou energia) das instruções Java.

Para aplicar este modelo, Mattos desenvolveu uma ferramenta que instrumenta os arquivos binários originais da aplicação Java (arquivos de extensão *class*) e gera um novo binário com métodos que acessam uma tabela de caracterização dos custos das instruções Java e acumulam o custo total da aplicação em uma variável que é impressa ao fim da execução, juntamente com um sumário ordenado por consumo apresentando todos os métodos que foram executados durante a execução da aplicação. É importante ressaltar que os métodos e classes inseridos durante a instrumentação do código binário originais não são contabilizados como custos de execução da aplicação.

Além dos custos de ciclos e energia a ferramenta DESEJOS também é capaz de informar dados sobre a utilização de memória feita pela aplicação, como, por exemplo, quais objetos foram instanciados durante a execução da aplicação e quais seus tipos. Quando utilizada para coleta de informação de ocupação de memória, a ferramenta também é capaz de informar o pico de consumo de memória da aplicação, facilitando a escolha de uma plataforma de hardware adequado para a execução da mesma.

A abordagem adotada por Mattos apresenta algumas características indesejáveis, como o fato de não considerar custos de um mecanismo de *garbage collector*, comum em máquinas virtuais Java. Além disso, a ferramenta realiza a coleta de dados através da inserção de *bytecodes* no arquivo binário da aplicação que se deseja obter as métricas físicas, podendo vir a inserir falhas na aplicação. Outro detalhe importante é o fato de que a ferramenta DESEJOS, em sua versão corrente, não é capaz de contabilizar os custos de chamadas a bibliotecas Java que tenham implementação em código nativo da máquina que executa a JVM. Também é importante ressaltar que pelo fato da ferramenta instrumentar a aplicação original, seu código também é executado sobre uma máquina virtual Java, o que pode levar a longos tempos de execução.

2.3 Correlações entre métricas de qualidade de software e métricas físicas

No contexto de sistemas de software complexos, principalmente os sistemas concebidos sob o Paradigma de Orientação a Objetos – POO, alguns trabalhos tem abordado a possibilidade de correlacionar as métricas de qualidade de software de uma aplicação com suas características físicas, como tempo de execução, consumo de energia ou ocupação de memória (REDIN, OLIVEIRA, et al., 2008), (MANGALAMPALLI e JAIN, 2007).

Redin et al. (2008) buscaram por correlações diretas entre as métricas de qualidade de software e as características físicas de uma determinada aplicação. Em seus experimentos foram analisadas diferentes implementações de duas aplicações: um decodificador de MP3 e um algoritmo de controle de uma cadeira de rodas, ambos implementados em linguagem Java[3]. As diferentes versões da mesma aplicação consistem de variações do nível de orientação a objetos nas implementações de uma mesma funcionalidade.

As métricas de qualidade de software utilizadas nestes experimentos foram às métricas que a ferramenta *Eclipse Metrics Plugin* (METRICS, 2005) é capaz de extrair, a descrição de algumas delas pode ser encontrada no Capítulo 3. Já as métricas físicas analisadas nos experimentos foram memória de programa, memória de dados, ciclos e energia.

Nos experimentos apresentados por Redin et al. (2008), as correlações estatísticas entre cada uma das métricas de qualidade de software e cada uma das métricas físicas das aplicações foram exploradas visando servir como guia para que os projetistas possam tomar decisões sobre a eficiência no uso desses recursos físicos ainda em etapas iniciais do projeto, baseado nas métricas de qualidade de software e nas correlações extraídas e apresentadas.

---

[3] Para a plataforma FemtoJava.

A metodologia adotada por Redin et al. (2008) apresenta alguns problemas, como o fato de a ferramenta *Eclipse Metrics Plugin Plugin* (METRICS, 2005) extrair métricas de todo o código do projeto, mesmo de bibliotecas ou classes que não são efetivamente utilizadas durante a execução da aplicação. Além disso, as correlações têm um caráter linear que pode não ser capaz de descrever adequadamente relacionamentos complexos entre as métricas de qualidade de software e as métricas físicas de uma determinada aplicação.

Mangalampalli e Jain (2007) também utilizaram correlações entre métricas de qualidade de software (métricas de alto nível) e características físicas (métricas de baixo nível) de aplicações orientadas a objetos. No entanto as correlações apresentadas neste trabalho não provem de um método estatístico e sim da análise das métricas (de alto e baixo nível) coletadas a partir das implementações. Essa classificação de correlações definida por Mangalampalli através da observação dos resultados contém 5 categorias de relação entre as métricas de alto nível e as de baixo nível, são elas: muito fraca, fraca, média, forte e muito forte.

Nos experimentos deste trabalho, Mangalampalli e Jain (2007) avaliaram três versões de uma mesma aplicação em linguagem C++, e a cada uma das versões eles atribuiram à classificação de muito, médio e pouco, de acordo com o quanto elas estão orientadas a objetos. Após, ele coletou as seguintes métricas relacionadas à arquitetura de herança das implementações, número de filhos (NOC), profundidade da árvore de herança (DIT), número de ancestrais (NOA), número de métodos herdados (NMI), número de métodos sobrescritos (NMO), medida de abstração de atributos (MAA) e medida de abstração funcional (MFA).

Por sua vez, as métricas físicas utilizadas no trabalho de Mangalampalli foram: tempo de execução e ocupação de memória. Essas características físicas foram extraídas através da ferramenta *Tunning and Analysis Utilities* (TAU) em um sistema Linux rodando sobre processadores, normalmente, encontrados em computadores pessoais e servidores (arquiteturas AMD64, IA32 e IA64).

A abordagem adotada por Mangalampalli e Jain é pouco interessante, pois não estabelece uma metodologia, bem embasada, a ser seguida, fazendo com que as correlações sejam muito dependentes da interpretação do que ele considerou uma aplicação muito ou pouco orientada a objetos.

2.4 MAISA (do inglês - *Metrics for Analysis and Improvement of Software Architectures*)

Verkano et al. (2000) propuseram uma ferramenta capaz de gerar predições de performance de modelagens em fases iniciais de projetos de software. Essas predições são baseadas em uma mineração de padrões de projeto[4] utilizados na aplicação.

Segundo os autores do referido trabalho, a detecção, preferencialmente automática, de padrões de projeto pode interferir positivamente no desempenho da aplicação, uma

[4]Padrões de projeto correspondem à descrição de soluções reusáveis e extensíveis para problemas recorrentes em projetos de sistemas de software (GAMMA, HELM, et al., 1995).

vez que se guarde na biblioteca de padrões de projeto um indicativo de seu custo em desempenho colhido de projetos anteriores. Associado ao armazenamento do custo histórico do padrão, os autores também defendem o uso de uma ferramenta capaz de realizar a detecção automática de padrões de projeto visto que ela seria capaz de detectar todos os padrões de projeto existentes na modelagem, mesmo aqueles que não foram inseridos intencionalmente.

Os autores afirmam que uma vez que um relatório (apresentando os padrões de projeto existentes na modelagem) esteja disponível e que o projetista tenha acesso a indicativos históricos do impacto destes padrões de projeto no desempenho de outras aplicações, seria possível detectar quais desses padrões podem ser antipadrões[5] no que se refere ao desempenho da modelagem em questão.

A abordagem de Verkano et al. propõe, então, uma ferramenta capaz de realizar a mineração de padrões de projeto sobre diagramas UML, além de gerar relatórios contendo informações sobre os padrões de projeto encontrados. Essa ferramenta recebeu o nome MAISA (do inglês *Metrics for Analysis and Improvement of Software Architectures*). Com base nesses relatórios e nos dados históricos de desempenho espera-se que o projetista infira quais padrões devem ser removidos do projeto antes mesmo de alcançar uma versão executável do produto de software. Segundo os autores esse esforço, ainda nas etapas iniciais do projeto para detecção de problemas se justifica, pois correções de erros detectados pelo usuário final geram custos em torno de uma ordem de grandeza mais altos.

Embora o referido trabalho tenha sido desenvolvido em conjunto com uma das maiores fabricantes de celulares do mundo, a finlandesa Nokia, as publicações científicas referidas no site do projeto[6] não apresentam dados quantitativos sobre a aplicação de sua metodologia, ainda que apresentem detalhadamente a abordagem proposta. Além disso, a ferramenta disponibilizada no site é dependente de uma biblioteca de padrões de projeto não divulgada, o que dificulta a utilização da mesma.

2.5 MILEPOST (do inglês *MachIne Learning for Embedded PrOgramS opTimization*)

Uma abordagem um pouco diferente, para estágios posteriores do processo de desenvolvimento, foi proposta por Fursin et al. (2008). Nessa abordagem os autores defendem o uso de métodos de aprendizado de máquina para tornar os compiladores mais inteligentes.

Fursin et al. propõem explorar as otimizações que surtirão melhores resultados em termos de tempo de execução e tamanho do código binário, ou ainda tempo de compilação. Além de selecionar quais otimizações usar, a ferramenta também sugere a melhor ordem para aplicação dessas otimizações selecionadas, dentre um conjunto pré-estabelecido durante a etapa de treinamento do algoritmo de aprendizado de máquina.

---

[5]Antipadrões são *padrões de projeto* que não se mostram adequadas em determinado contexto, pois resolvem o problema ao qual se aplicam, mas geram outros problemas em decorrência de sua aplicação (KOENIG, 1995).

[6]http://www.cs.helsinki.fi/group/maisa/

Como um passo intermediário nesse trabalho, foi necessário desenvolver uma Interface Interativa de Compilação (ICI – inglês *Interactive Compilation Interface*) (FURSIN e COHEN, 2007) para o compilador *GNU Compiler Collection* (GCC), tornando possível a escolha de quais otimizações aplicar, bem como em que ordem essas otimizações seriam executadas no processo de compilação da aplicação. Essas ações de otimização, usualmente, são tomadas através do parâmetro "-O", seguido de um número indicativo do esforço de otimização (variando de 0 a 3). Nos resultados obtidos por Fursin et al. fica claro que a abordagem original do GCC não é ótima.

Em seus experimentos, Fursin et al. alcançaram resultados que demonstraram que a simples manipulação da ordem em que as otimizações, já existentes no GCC, seriam executadas foi capaz de causar uma aceleração de 2,31 vezes em comparação a uma versão da mesma aplicação compilada pelo GCC original através do mecanismo de otimização acionado pelo parâmetro "-O3".

Na metodologia proposta por Fursin et al., a tomada de decisões é baseada em características estruturais, obtidas do código fonte da aplicação a qual se deseja otimizar. Em um estágio anterior a compilação são coletadas 55 características estáticas do código fonte da aplicação, dentre elas: número de blocos básicos, números de constantes, média do número de instruções em um bloco básico, etc.

O conjunto de treinamento do método de aprendizado de máquina foi formado por uma série de associações entre as características estruturais, as características físicas (tempo de execução, tamanho do binário, etc.) e o arranjo de *flags* de otimização utilizadas para a compilação da aplicação. Em outras palavras, a mesma aplicação é compilada diversas vezes, variando-se os perfis de otimização e coletando-se as características físicas que se deseja controlar. Para uma mesma aplicação, só são apresentados ao método de aprendizado de máquina os exemplos de treinamento que alcancem ao menos 98% da aceleração do exemplo que alcançou a maior aceleração.

Para estimar qual o melhor perfil de otimização para uma determinada aplicação o método utilizado por Fursin et al. foi o método *k-nearst-neighbors* (MITCHELL, 1997), assumindo k=1, ou seja, o mecanismo de decisão decide qual das aplicações apresentadas durante o treinamento está mais próximo da nova aplicação no espaço de características e retorna o conjunto de *flags* de otimização que melhor se enquadre as necessidades do usuário (menor tamanho de código binário, menor tempo de execução, ou ambos).

O conjunto de *benchmarks* MiBench (GUTHAUS, RINGENBERG, *et al.*, 2001) foi utilizado para gerar o conjunto de treinamento do método de aprendizado de máquina. Para tal, cada aplicação foi compilada com 500 conjuntos diferentes de *flags* de otimização (ordenadas aleatoriamente) e foi executada 5 vezes para reduzir influências do sistema operacional sobre os resultados.

Além disso, para os testes foi utilizada uma abordagem *leave-one-out-cross-validation* (MITCHELL, 1997), onde a aplicação a ser testada foi retirada do conjunto de treinamento do método de aprendizado de máquina, assim sendo, os resultados de predição não são viciados.

Ainda, os autores destacam que para a execução desses experimentos os dados de entrada dos *benchmarks* foram retiradas do conjunto de dados disponibilizados no

projeto MiDataSets[7] (FURSIN, CAVAZOS, et al., 2007), utilizando sempre os conjuntos de dados rotulados como “No1”.

Os resultados da escolha automática de perfis de otimização mostrou-se bastante interessante, visto que aplicações como *blowfish*, *susan* e *gsm* obtiveram acelerações em torno de 2 vezes (de 1,97 a 2,31). No entanto, as arquiteturas escolhidas para os experimentos não pertencem ao contexto de sistemas embarcados. Além disso, embora a proposta seja de um sistema para a otimização de diferentes características físicas, inclusive de forma associada, o trabalho só apresenta resultados para tempo de execução.

Atualmente, as ferramentas MILEPOST-GCC e ICI foram integradas em um projeto maior denominado *cTuning Compiler Collection* (*cTuning* CC)[8].

[7]http://midatasets.sourceforge.net

[8]http://ctuning.org

3 MÉTRICAS DE SOFTWARE

Há mais de um século, o físico-matemático britânico William Thomson[9] afirmou que a capacidade de mensurar e expressar algo em números permite um conhecimento mais profundo sobre determinado assunto. Em outras palavras, ele postulou que a incapacidade de expressar qualquer entidade em números é prova fidedigna de um conhecimento pobre e insatisfatório acerca dessa entidade.

A constatação do grande cientista irlandês foi apresentada por Möller e Paulish, em seu livro "*Software Metrics*" (MÖLLER e PAULISH, 1992), com o intuito de evidenciar o fato de que há muito tempo é sabido que nós entendemos de forma mais clara aquilo que podemos mensurar. Embora as afirmações do referido Barão britânico terem sido apresentadas em trabalhos relacionados à análise de fenômenos físicos, elas são facilmente aceitas como um conceito mais geral, onde pode-se interpretar processos como fenômenos. O trabalho de Kelvin pode ser visto em prática, ainda hoje, nas técnicas que visam a melhorar a qualidade e a produtividade de processos através da aplicação de métodos quantitativos.

Esses métodos quantitativos são baseados em medições. Uma medição é a atribuição de números a objetos ou eventos de acordo com uma regra. Essa regra de atribuição pode ser qualquer regra consistente, desde que não aleatória (KANER e BOND, 2004). Ainda, segundo Fenton e Pfleeger (1996), uma medição pode ser definida formalmente como o mapeamento do mundo empírico para o mundo formal. Consequentemente, uma medida é o número ou símbolo atribuído a uma entidade por este mapeamento de forma a caracterizar um atributo.

Um dos objetivos básicos da engenharia de software é transformar a atividade de criação de software de um processo artístico, pobremente entendido e indisciplinado em uma atividade cuidadosamente controlada, metódica e previsível (MÖLLER e PAULISH, 1992). Portanto, para tornar o produto de software algo que possa ser projetado é importante que os projetistas, desenvolvedores e "mantenedores" sejam capazes de expressar as características do sistema em termos quantitativos. A aplicação de métricas no domínio da engenharia de software começou em meados da década de 1970, como uma ferramenta para a gerência efetiva dos processos de desenvolvimento e manutenção, sendo usadas métricas para medir e avaliar a qualidade do produto de software (MÖLLER e PAULISH, 1992).

---

[9] 1º Barão de Kelvin, no Brasil, mais conhecido como Lorde Kelvin, criador da escala absoluta de temperaturas.

A medição quantitativa é comumente usada para avaliar algumas características de software, como por exemplo, o desempenho do software pode ser medida quantitativamente em termos de tempo de execução, tempo de resposta, uso de memória, etc. (LYNCH e BROWNE, 1981). Outro exemplo é a medição quantitativa da confiabilidade do software em termos de tempo médio entre falhas, tempo médio para correção de um erro ou densidade de erros (MUSA, 1975).

Com relação à qualidade de software, muitos fatores devem ser considerados. Alguns destes fatores, tais como confiabilidade e eficiência, medem atributos do produto de software. Estas métricas são chamadas de métricas do produto de software. Outros tantos fatores, no entanto, referem-se ao processo pelo qual o sistema foi construído e medidas destes atributos são chamadas métricas do processo de desenvolvimento de software (KAFURA, 1985).

Muitos ambientes têm sido propostos para avaliar a qualidade de produtos de software pela comunidade de engenharia de software, exemplos destas propostas podem ser encontrados em Hendersen-Sellers (1996) e Martin (2002). Dentre a bibliografia, um trabalho se destaca no estudo e revisão de métricas para avaliação de software orientado a objetos (XENOS, STAVRINOUDIS, *et al.*, 2000). Recentemente, alguns pesquisadores aplicaram um estudo empírico das métricas de orientação a objetos em três implementações de uma mesma aplicação e estudaram suas relações com os consumos de memória e processamento (MANGALAMPALLI e JAIN, 2007).

Como, geralmente, as métricas de engenharia de software são aplicadas com o objetivo de medir e avaliar a qualidade do software faz-se então necessária a apresentação de uma definição para qualidade. A seguir é apresentada uma definição recorrente, extraída da Norma NBR 8402 da ABNT/ISO (ABNT, 1994), para o termo qualidade. “Qualidade é a totalidade das características de um produto ou serviço que se refletem na sua capacidade de satisfazer necessidades especificadas ou implícitas”.

Essa definição de qualidade por si só não é muito útil para o desenvolvedor do sistema de software. O interesse deste foca-se em saber como desenvolver produtos de software com “boa” qualidade. Por isto, faz-se necessário que ele identifique propriedades físicas ou abstratas (atributos) de qualidade do software cuja presença ou ausência possa ser reconhecida. As métricas são ferramentas que ajudam a quantificar estes atributos de qualidade, de forma que se possa medir o efeito das ações realizadas para melhorar a qualidade do software.

Como visto anteriormente, as métricas podem medir inúmeras características do produto de software e até mesmo do processo de desenvolvimento do produto de software. Neste trabalho, o foco estará nas métricas de software voltadas ao produto de software. Maiores informações sobre métricas do processo de desenvolvimento podem ser obtidas em outras fontes da literatura ((IEEE, 1998), (MÖLLER e PAULISH, 1992), (KAFURA, 1985)). Além disso, o Apêndice B apresenta um glossário extraído do padrão IEEE para métricas de qualidade de software (IEEE, 1998).

3.1 Origens

O fundamento para as origens da aplicação de métodos quantitativos para o desenvolvimento de software se estabeleceu em meados da década de 1970. Ainda assim, as métricas de software continuavam sendo uma disciplina tida como nova no início da década de 1990 (MÖLLER e PAULISH, 1992).

Segundo Möller e Paulish, a origem das métricas de software provem de quatro tendências básicas que foram sendo aperfeiçoadas e evoluíram para as métricas de engenharia de software (MÖLLER e PAULISH, 1992). São elas: Medidas de Complexidade do Código, Estimação do custo do Projeto de Software, Garantia de Qualidade de Software e Processo de Desenvolvimento de Software.

Para medidas de complexidade de código foram definidas métricas tais como Complexidade Ciclomática de McCabe (MCCABE, 1976) e Ciência de Software de Halstead (HALSTEAD, 1977) que podem ser facilmente obtidas a partir do código fonte, inclusive através de métodos automatizados.

Da mesma forma, foram definidas medidas para estimação do custo de projetos, que permitem avaliar o esforço e tempo requeridos para que o produto de software seja desenvolvido, em geral, baseiam-se na estimativa de número de linhas de código necessárias para a implementação do projeto, além de outros fatores relacionados a complexidade do projeto. Os primeiros exemplos de técnicas deste tipo incluem Modelo SLIM de Larry Putnam (PUTNAM, 1978) e o Modelo COCOMO de Barry Boehm (BOEHM, 1981). Visando a garantia da qualidade de software, dados de falhas do produto de software ocorridas durante várias fases do ciclo de vida do software também eram coletados.

Com o aumento do tamanho e da complexidade dos projetos de software emergiu a necessidade de um processo controlado de desenvolvimento de software. Este processo incluía a definição do ciclo de vida do software como uma sequência finita de etapas, e enfatizava o gerenciamento do projeto com um melhor controle dos recursos. Neste contexto, as medidas de recursos e custos de desenvolvimento resultantes eram coletadas usando sistemas corporativos de contabilidade de custos.

3.2 Métricas do Produto de Software

As métricas são importantes em qualquer processo que envolva engenharia. Muitas métricas têm sido propostas para avaliar a qualidade de produtos e de processo de software. Aqui algumas métricas do produto de software relacionadas a este trabalho serão apresentadas. Cada métrica será apresentada com uma breve descrição textual explicando sua aplicação. Além disso, exemplos gráficos e formalizações matemáticas serão apresentados quando possível.

É importante ressaltar que embora as métricas devam ser definidas de forma a evitar ambiguidades em sua interpretação, algumas vezes encontramos na bibliografia formas distintas, no entanto homônimas, de mensurar um dado atributo do software. Isso acontece ora por escolha do projetista da ferramenta de extração de métricas, ora por necessidade devido ao nível de abstração utilizado na representação do software. Por exemplo, a métrica WMC (do inglês *Weigthed Methods per Class*) representa a classe como um somatório de complexidades de seus métodos. Essas complexidades são representadas por vezes como a complexidade ciclomática do método (comumente conhecida como *VG*), como no *Eclipse Metrics Plugin* (METRICS, 2005), outras vezes como o número de linhas do método (MLOC) e até mesmo como uma medida da complexidade dos parâmetros do método, por exemplo, como ocorre no editor de diagramas UML MagicDraw[10]. Esse último caso é especialmente interessante, pois

[10]http://www.magicdraw.com/

adéqua a métrica a um nível de abstração mais alto onde não seria possível como utilizar a implementação mais comum da métrica, uma vez que *MLOC* e *VG* não poderiam ser obtidas se não houver código referente aos métodos.

Nas descrições a seguir serão adotadas as notações mais usuais para o cálculo das métricas apresentadas.

3.2.1 Número de Classes (*NC – do inglês Number of Classes*)

A métrica número total de classes, ou simplesmente número de classes, representa a contagem de classes, abstratas e concretas, em um escopo selecionado.

Não existem intervalos ou limites sugeridos para os valores desta métrica. O número de classes de um projeto de software dependerá das necessidades específicas de cada projeto e da aplicação adequada, ou não, dos conceitos de desenvolvimento orientado a objetos.

3.2.2 Número de Interfaces (*NI – do inglês Number of Interfaces*)

Uma Interface é uma ferramenta capaz de criar uma camada extra de abstração em projetos orientados a objetos. Na linguagem de programação Java, por exemplo, uma interface define as assinaturas e constantes de uma espécie de classe abstrata onde detalhes de implementação não são permitidos. Classes relacionadas à interface deverão implementar os métodos definidos na interface. A métrica número de interfaces consiste da contagem de interfaces definidas no escopo selecionado para análise.

3.2.3 Número de Pacotes (*NOPK – do inglês Number of Packages*)

Um pacote é um grupo de classes relacionadas e possivelmente cooperantes. Os pacotes facilitam o reúso e ajudam a resolver problemas de ambiguidade para classes com o mesmo nome. Em linguagem de programação Java os pacotes podem ser arranjados de forma hierárquica, restringindo os escopos quando necessário.

A métrica número de pacotes é definida como a contagem do número de pacotes existentes no escopo selecionado para análise, incluindo eventuais subpacotes.

3.2.4 Número de Parâmetros (*NP – do inglês Number of Parameters*)

A métrica número de parâmetros representa uma medida de complexidade da associação entre objetos, visto que métodos com maior número de parâmetros representam um acoplamento maior entre a classe chamada e a classe chamadora (BALASUBRAMANIAN, 1996), e, como será visto posteriormente, o acoplamento é uma característica que deve ser reduzida.

A medida NP de um método consiste da contagem dos parâmetros a serem recebidos em uma chamada do método em questão. A Figura 3.1 apresenta a classe Ponto, onde, em destaque, os parâmetros dos métodos *setX*, *setY* e *setZ* podem ser vistos. Cada um destes métodos tem um único parâmetro, portanto, a métrica *NP* é igual a um para os referidos métodos.

Ainda na Figura 3.1, os métodos *getX*, *getY* e *getZ* apresentam NP igual a zero, visto que não possuem parâmetros.

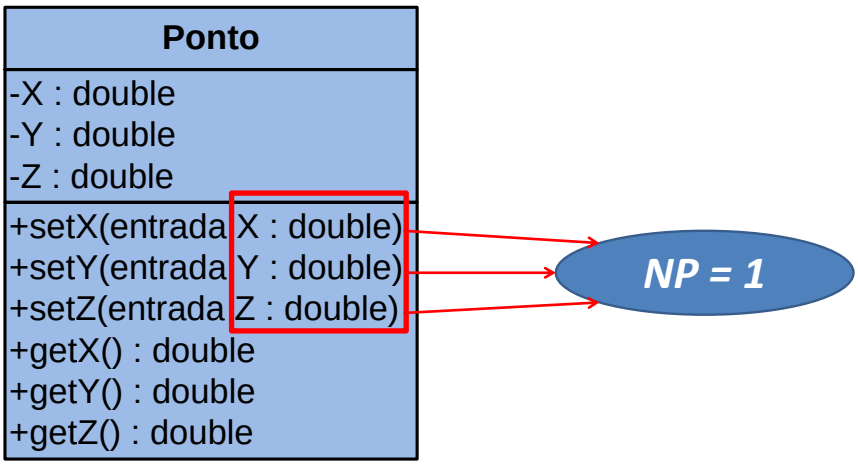

Figura 3.1: Exemplo: Número de Parâmetros.

3.2.5 Número de Métodos (*NOM – do inglês Number of Methods*)

Em projetos onde o paradigma orientado a objetos é utilizado, os métodos, ou operações, são os componentes de uma classe responsáveis por descrever seu comportamento (SOMMERVILLE, 2006). A contagem de métodos de uma classe é uma forma simples de medida de complexidade. De acordo com Chidamber e Kemerer (1994), classes com muitos métodos tendem a causar grande impacto em suas subclasses, uma vez que seus descendentes herdam seus métodos. Porém, classes com poucos métodos podem indicar um problema de projeto, ou seja, talvez seus atributos e métodos não sejam suficientes para constituir um novo tipo ou conceito (PRESSMAN, 2002).

A métrica número de métodos, ou número total de métodos, é facilmente obtida, visto que seu cálculo consiste em contar o número de métodos presentes na classe, sejam eles concretos, abstratos, estáticos, públicos, protegidos ou privados. O número total de métodos da classe Ponto, apresentada na Figura 3.2, é seis (*NOM = 6*). Na referida figura as setas destacam os métodos representados em uma classe em conformidade com a UML.

Além do número total de métodos, algumas ferramentas também medem o número de métodos estáticos, que também são conhecidos como métodos de classe, em uma classe. Essa métrica pode dar informações sobre uso de operações sobre a classe como um todo, não apenas em suas instâncias.

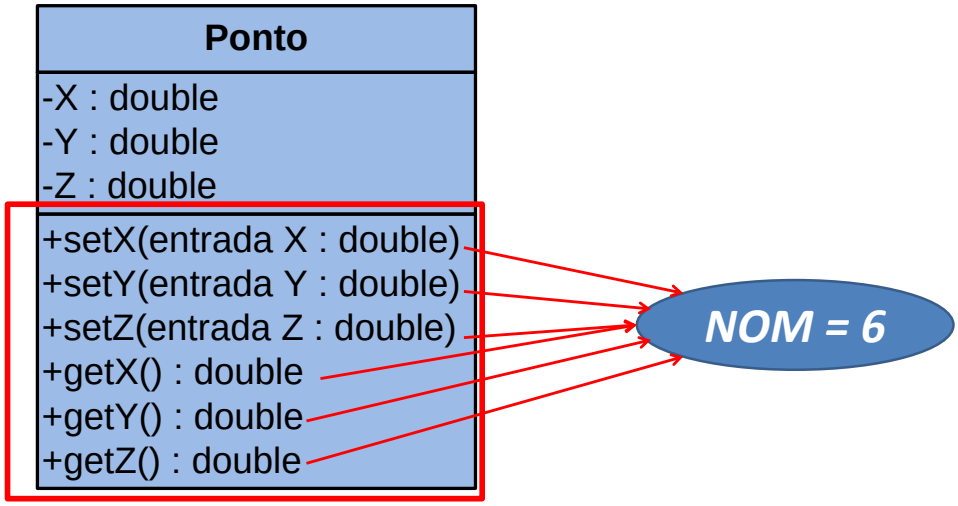

Figura 3.2: Exemplo: Número de Métodos de uma classe.

3.2.6 Número de Campos (*NF – do inglês Number of Fields*)

A métrica número de campos, ou número de atributos, representa a quantidade de atributos de uma classe. Os atributos são os componentes da classe responsáveis por armazenar seu estado (PRESSMAN, 2002).

Embora uma grande quantidade de campos em uma classe não seja, necessariamente, uma indicação de projeto ou código ruim, isto pode indicar que uma nova classe poderia ser extraída, de forma a agrupar alguns desses atributos. É importante ressaltar que aqueles métodos que atuam sobre os atributos selecionados devem, também, ser movidos para a classe extraída. A extração de classes pode melhorar a semântica do modelo da aplicação e gerar uma melhor distribuição da responsabilidade do sistema (PRESSMAN, 2002), além de aumentar as possibilidades de reuso das classes do projeto.

Algumas ferramentas, além da extração do número de campos, também coletam o número de campos estáticos, ou seja, os atributos da classe que foram definidos estaticamente e serão compartilhados entre todas as instâncias da classe.

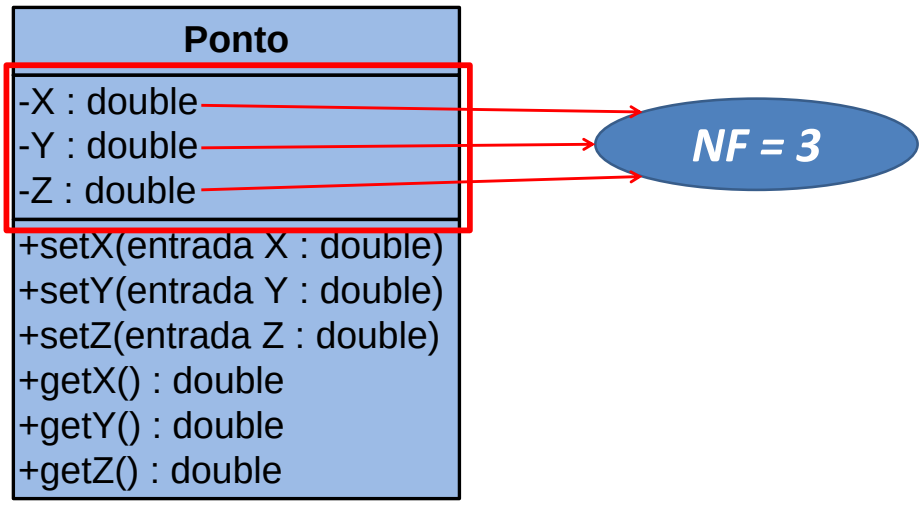

Figura 3.3: Exemplo: Número de Campos.

Na Figura 3.3 é ilustrada a mesma classe da Figura 3.2, a classe Ponto, agora destacando os atributos da classe. A medida Número de Atributos ou Campos, NF, para esta classe é três.

3.2.7 Profundidade de Blocos Aninhados (*NBD – do inglês Nested Block Depth*)

A métrica profundidade de blocos aninhados representa o número máximo de blocos de código aninhados (*Nested Block Depth*) em um determinado método de uma classe. Essa métrica é um indicador de complexidade de módulo, no caso de um método. Os comandos de controle da linguagem (como IF, ELSE, FOR, FOREACH, WHILE), quando aninhados em excesso, além de tornarem a execução mais complexa, dificultam a compreensão e a legibilidade do código fonte, e com isso sua manutebilidade (SOMMERVILLE, 2006). Métodos que apresentam a métrica NBD acima de seis necessitam de refatoração para facilitar a sua manutenibilidade.

A Figura 3.4 apresenta a definição de uma classe com um método em linguagem Java. Na mesma figura estão destacados os dois blocos que possuem a profundidade máxima de blocos aninhados do método *main*. Como definido anteriormente, a métrica *NBD* é dada pela profundidade do bloco mais aninhado no método, portanto o método *main* possui *NBD = 2*.

```
public class ClasseExemplo {
    public static void main(String args[]) {
        double a, b, x = 1.5, erro = 0.05;
        a = 1;
        b = 2; // 1 < (raiz de 2) < 2
        while ((b - a) > erro) {
            x = (a + b) / 2;
            if (x * x < 2) // x < raiz de 2
                a = x;
            else
                // x >= raiz de 2
                b = x;
        }
        System.out.println("Valor aproximado de raiz quadrada de 2: " + x);
    }
}
```

NBD = 2

NBD = 2

Figura 3.4: Exemplo de aplicação da métrica NBD.

3.2.8 Linhas de Código (*LOC – do inglês Lines of Code*)

Talvez o fato mais importante sobre a métrica *Linhas de Código* seja entender que embora seu cálculo aparentemente seja extremamente simples, na verdade ele exige a interpretação dos dados com um certo nível de semântica.

Quando a programação era feita em *assembly* e cada linha representava uma instrução do processador, a contagem de linhas era trivial. Já nos tempos atuais, com a presença massiva de linguagens de alto nível de abstração nos projetos de software, essa antiga correspondência um para um se perdeu, e uma linha de código pode resultar em várias instruções. Diferentes linguagens de programação, ou mesmo diferentes comandos da mesma linguagem de programação podem levar a grandes variações nessa métrica.

Para demonstrar essas variações a Figura 3.5 apresenta a comparação de três trechos de código em linguagem Java, todos com mesma semântica, porém utilizando diferentes comandos da linguagem e apresentando diferentes números de linhas de código.

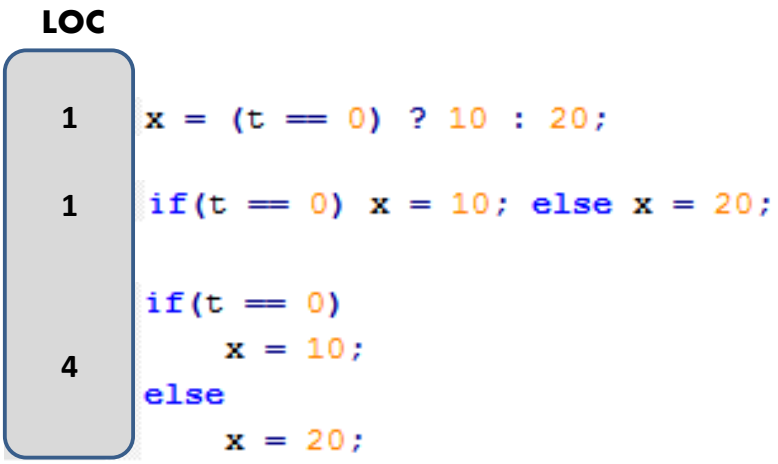

Figura 3.5: Trechos de código com equivalência semântica.

Jones (1986) discute a complexidade inerente ao processo de contagem das linhas de código e descreve diversas variações no modo de realização dessa contagem, entre elas:

- contar todas as linhas executáveis;
- contar as linhas executáveis juntamente com as linhas de declaração de dados;
- contar as linhas executáveis juntamente com as linhas de declaração de dados e linhas de comentários;
- contar as linhas executáveis juntamente com as linhas de declaração de dados, linhas de comentários e comandos de controle da linguagem;
- contar todas as linhas;
- contar as linhas de blocos de código entre terminadores lógicos.

Para explicitar melhor as variações na aplicação dessa métrica apresentar-se-ão alguns exemplos presentes na literatura. Boehm (1981) sugere a contagem de linhas de código como sendo a soma de todas as linhas que contenham comandos executáveis da linguagem, bem como as linhas contendo declarações de dados e comentários.

Por sua vez, Conte, Dunsmore e Shen (1986) e propõe que a contagem de linhas de código seja calculada como o somatório de todas as linhas que não sejam linhas de comentário ou linhas em branco, isso inclui todas as linhas de declarações de dados e cabeçalhos de programa, bem como todos os elementos da linguagem, executáveis ou não. Assim sendo, essa metodologia de contagem de linhas de código inclui preâmbulos e declarações de dados, mas exclui linhas de comentários. Já o método proposto pela IBM[11] de Rochester é que a contagem de linhas de código inclua as linhas executáveis e de declaração de dados, mas exclua as linhas de preâmbulo e comentários (KAN, 2002).

Atualmente, devido ao crescente tamanho dos projetos de software, essa métrica tem aparecido renomeada como KLOC (do inglês *Kilo Lines of Code*) ou MLOC (do inglês

[11]IBM - *International Business Machines.*

Million Lines of Code) indicando, a contagem em milhares ou milhões de linhas de código (MÖLLER e PAULISH, 1992). No entanto, é importante ressaltar que as mesmas metodologias de contagem são aplicadas a essas variações da métrica, tendo como única diferença o multiplicador indicativo de uma maior ordem de grandeza nos dados levantados.

Além das variações relativas à interpretação da semântica das linhas de código, discutidas acima, atualmente existem distinções na implementação dessa métrica quanto ao escopo a ser avaliado. Basicamente, em implementações orientadas a objetos essa métrica é, geralmente, processada tanto para o escopo global do projeto quanto para métodos isolados.

A Figura 3.6 apresenta um fragmento de código em linguagem de programação Java, que representa a declaração de uma classe, bem como a contagem da métrica TLOC (do inglês *Total Lines of Code*) para essa classe. Na mesma figura pode-se ver o número de linhas físicas do arquivo, denotado por PLOC (do inglês *Physical Lines of Code*), e pode-se verificar a diferenciação de ambas as contagens.

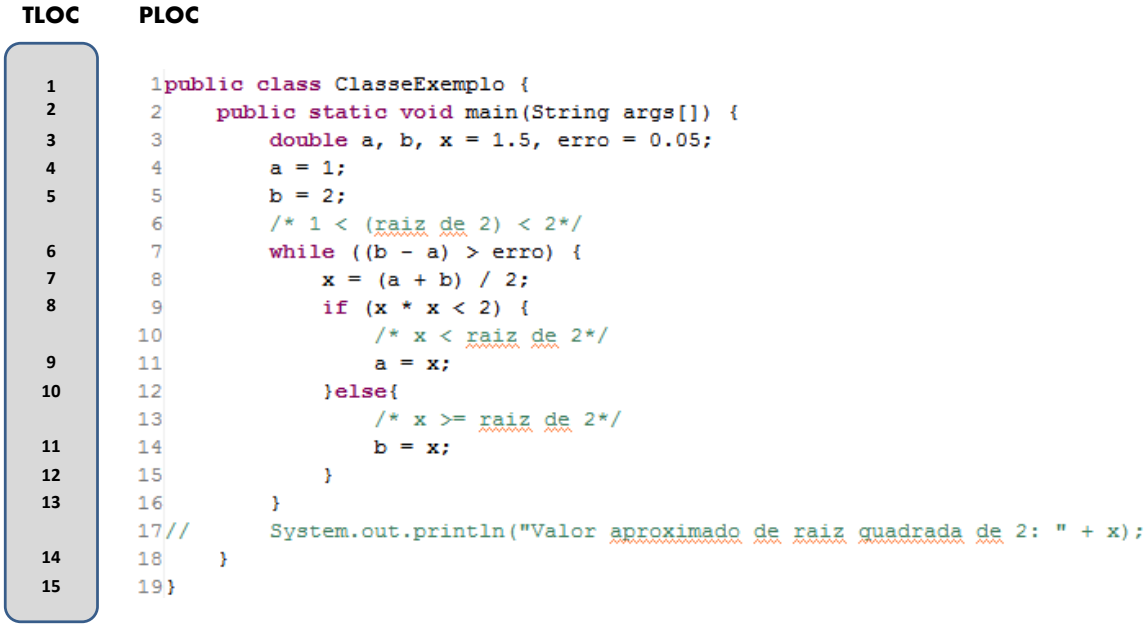

Figura 3.6: Exemplo de aplicação da métrica Linhas de Código

A Figura 3.7 apresenta um fragmento de código em linguagem de programação Java, que representa a declaração de um método, bem como a contagem da métrica *Method Lines of Code* para esse método. Para fins de desambiguação declara-se que neste trabalho MLOC remete a *Method Lines of Code*, não a *Million Lines of Code*.

```
LOC
      /// <summary>
      ///     Parses the specified command-line arguments.
      /// </summary>
      bool ParseCommandLine(string[] arguments)
      {
          // By default we show help if no
          // command-line arguments are specified
1         if (arguments.Length == 0)
          {
2             ShowHelp();
3             return false;
          }

4         for (int i = 0; i < arguments.Length; i++)
          {
5             if (arguments[i] == "/?")
              {
6                 ShowHelp();
7                 return false;
              }

8             if (arguments[i] == "/input")
              {
                  // We only recognize the /input
                  // switch, if the file name exists
9                 if (arguments.Length > 1 && File.Exists(arguments[i + 1]))
                  {
10                    InputFileName = arguments[i++];
                  }
              }
          }

11        return true;
      }
```

Figura 3.7: Exemplo de aplicação de Número de Linhas de Código do Método.

É importante ressaltar que a contagem de linhas de código, em geral, é utilizada como fator de ponderação para o cálculo de outras métricas, como, por exemplo, taxa de defeitos por número de linhas de código (MÖLLER e PAULISH, 1992).

Em nossos experimentos, as métricas MLOC e TLOC são utilizadas, sendo importante ressaltar que a ferramenta *Eclipse Metrics Plugin* (METRICS, 2005) realiza a contagem de linhas de código considerando todas as linhas físicas contidas no arquivo de fonte, excetuando-se as linhas que obedeçam à semântica de comentários da linguagem de programação Java e as linhas em branco.

3.2.9 Complexidade Ciclomática (CC ou V(G)[12])

A medida de *Complexidade Ciclomática*, proposta por Thomas McCabe (MCCABE, 1976), foi desenvolvida para expressar as capacidades de teste e manutebilidade do código fonte de um dado programa. McCabe concebeu-a como o número ciclomático da teórica clássica dos grafos, que por sua vez representa a quantidade de regiões em um grafo (KAN, 2002). Este trabalho não tem o intuito de discutir a teoria de grafos que dá embasamento a tal conceito, mas maiores informações a respeito da Teoria de Euler para Poliedros, de onde derivam esses conceitos, podem ser obtidas em Lima (1985).

[12]Em referência à teoria clássica de grafos que deu origem a métrica.

Na abordagem proposta por McCabe os programas são vistos como grafos dirigidos, planares, com um nodo de entrada e pelo menos um nodo de saída. Nesse modelo o grafo deve possuir caminhos que permitam que todo o nodo seja alcançável a partir do nodo de entrada. Além disso, pra todo nodo do grafo deve existir ao menos um caminho que leve a um nodo de saída. Esses grafos são conhecidos como *Grafos de Fluxo de Controle* (LEDGARD e MARCOTTY, 1975).

3.2.9.1 Estrutura básica de Grafos de Fluxo de Controle

Em um grafo de fluxo de controle cada nó representa um bloco básico do código do programa e as arestas representam o caminho lógico entre os blocos básicos. Em geral os nodos têm uma saída, no entanto os comandos de controle da linguagem de programação são convertidos em um tipo especial de nodo chamado de predicativo. Isso ocorre porque eles representam decisões e podem gerar diferentes caminhos no fluxo de execução da aplicação, por isso esses nós tem duas ou mais arestas de saída.

A Figura 3.8 apresenta a representação de dois blocos básicos em sequência na forma de um grafo de fluxo de controle simples com o fluxo de controle partindo do bloco básico A para o bloco básico B.

Figura 3.8: Exemplo de sequência em um grafo de fluxo de controle.

A Figura 3.9 apresenta a representação de uma estrutura Se-Então-Senão em um grafo de fluxo de controle, onde o nó A é um nó predicativo. Algumas variações da métrica de complexidade ciclomática propõem, que, quando mais de um predicado booleano estiver contido em um mesmo comando de seleção, a complexidade deve ser incrementada em uma unidade para cada predicado extra. Esta forma de computar a complexidade ciclomática é frequentemente referida como complexidade ciclomática estendida, ou CC2 (ZUSE, 1991). Vale ressaltar que o próprio McCabe sugeria que comandos de seleção com mais de um predicado lógico fossem representados como vários nós predicativos, tantos quantos fossem necessários (MCCABE, 1976).

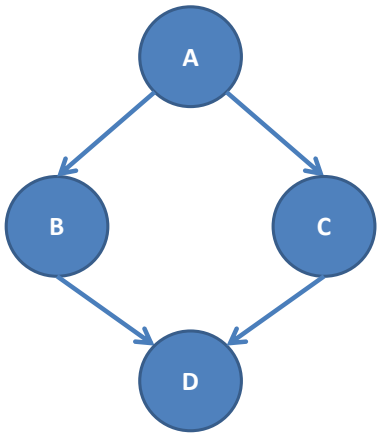

Figura 3.9: Exemplo de representação de uma estrutura Se-Então-Senão em um grafo de fluxo de controle.

A Figura 3.10 representa um laço com validação final (do tipo Repita-Até) onde o nodo B é predicativo e realiza a avaliação do predicado podendo gerar uma mudança do

fluxo de controle para o bloco interno do laço (A), ou para o nodo C caso o predicado não seja atendido.

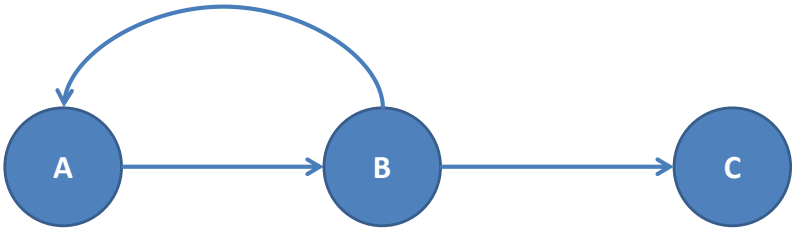

Figura 3.10: Exemplo de laço de repetição com validação final em um grafo de fluxo de controle.

A Figura 3.11 apresenta a representação de um laço com validação inicial, do tipo Enquanto, onde o nodo A é predicativo e realiza a validação, podendo gerar um deslocamento do fluxo de controle para o bloco interno do laço (B) ou para o nodo C, caso o predicado do nó A não seja atendido.

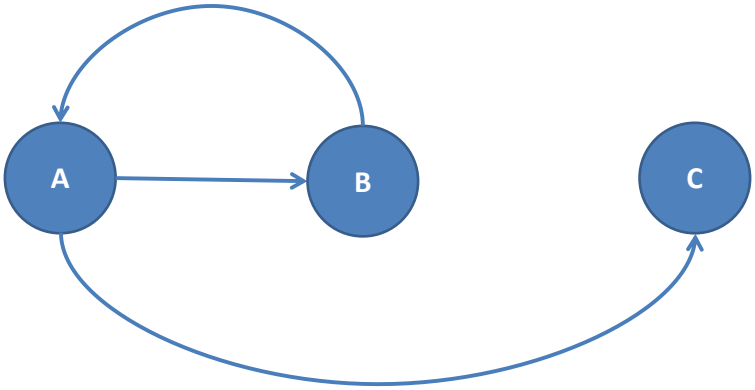

Figura 3.11: Exemplo de representação de um laço de repetição com validação inicial em um grafo de fluxo de controle.

A representação de um comando de seleção múltipla (Switch-Case) é apresentada na Figura 3.12. O nó predicativo A pode gerar desvios de execução para qualquer um dos nós B, C, .., D e após a execução de cada um desses nós o controle do programa passa para o bloco E.

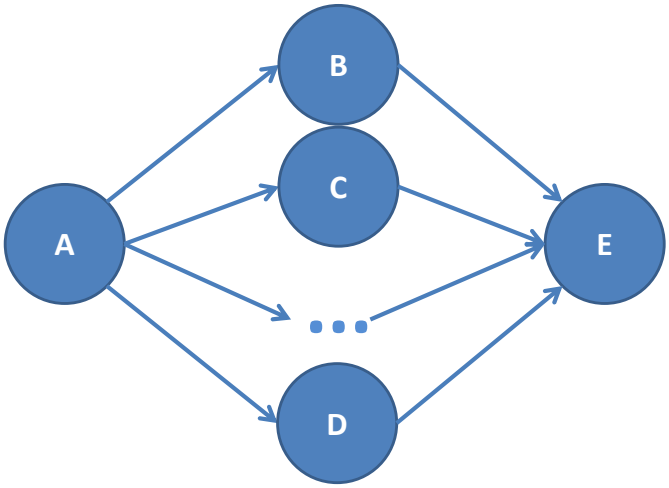

Figura 3.12: Exemplo de representação de um comando de seleção múltipla em um grafo de fluxo de controle.

3.2.9.2 O Número Ciclomático de um Grafo de Fluxo de Controle

O número ciclomático de um grafo G, ou a complexidade ciclomática de um programa representado pelo grafo de fluxo de controle G, pode ser obtido pelas fórmulas a seguir:

$$V(G) = e - n + 2 \cdot p \qquad \text{Equação 3.1}$$

$$V(G) = n_p + 1 \qquad \text{Equação 3.2}$$

$$V(G) = n_r \qquad \text{Equação 3.3}$$

onde:

G = grafo de fluxo de controle representativo do programa;

V(G) = valor indicativo da complexidade de G;

e = número de arestas de G;

n = número de nodos de G;

n_p = número de nós predicativos em G;

n_r = número de regiões distintas em G;

p = número de componentes conectados no grafo G.

A Figura 3.13 apresenta uma definição de classe em linguagem Java e código referente a seu método *main* e a Figura 3.14 apresenta o grafo de fluxo de controle do método *main* desta classe. Ainda, a partir do grafo apresentado na Figura 3.14 pode-se calcular a complexidade ciclomática do método por ele representado. Segundo as fórmulas anteriores a complexidade ciclomática do método *main* da classe apresentada na Figura 3.13 pode ser calculada da seguinte maneira:

e = número de arestas = 11;

n = número de nodos = 10;

n_p = número de nós predicativos = 2;

n_r= número de regiões = 3;

p = 1.

$$V(G) = e - n + 2 \cdot p = 11 - 10 + 2 \cdot 1 = 3$$

$$V(G) = n_p + 1 = 2 + 1 = 3$$

$$V(G) = n_r = 3$$

```java
public class ClasseExemplo {
    public static void main(String args[]) {
        double a, b, x = 1.5, erro = 0.05;
        a = 1;
        b = 2; // 1 < (raiz de 2) < 2
        while ((b - a) > erro) {
            x = (a + b) / 2;
            if (x * x < 2) // x < raiz de 2
                a = x;
            else
                // x >= raiz de 2
                b = x;
        }
        System.out.println("Valor aproximado de raiz quadrada de 2: " + x);
    }
}
```

Figura 3.13: Código fonte de uma classe em linguagem Java.

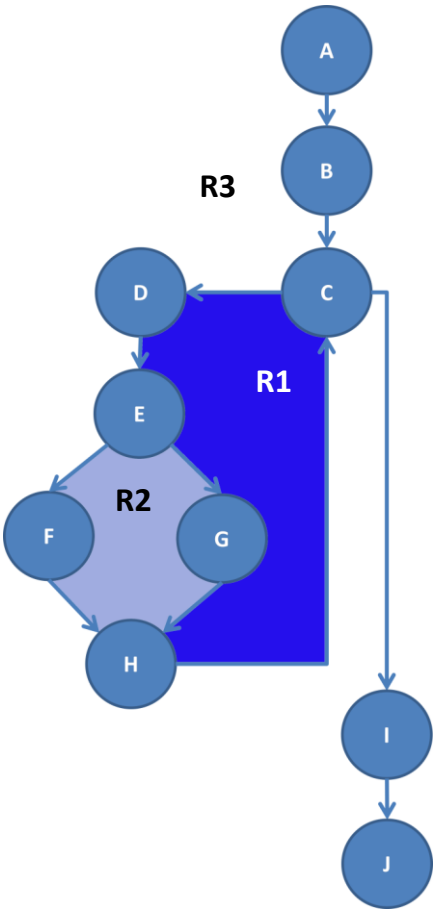

Figura 3.14: Exemplo de grafo de fluxo de controle para a para o método *main* apresentado na Figura 3.13.

Um fato interessante, e geralmente relegado em outros trabalhos, sobre a complexidade ciclomática é a presença do coeficiente p na fórmula original. O conceito de componentes pode parecer ir contra o conceito básico de grafo de fluxo de controle definido anteriormente (onde cada nó deve ser acessível pelo nó de entrada e os nós de saída devem ser acessíveis por todos os nós). Isso porque o conceito de grafo de fluxo de controle por si só define um grafo com um único componente ($p = 1$). No entanto,

McCabe propõe que sub-rotinas podem ter seus próprios grafos de controle, e que esses grafos também são componentes do grafo de controle da aplicação (MCCABE, 1976). A Figura 3.15 apresenta um exemplo de grafo de fluxo de controle onde o fluxo de controle principal (denotado por M) tem duas sub-rotinas (denotadas por A e B). Nesse exemplo, $p = 3$ e, portanto, $V(G) = e - n + 2 \cdot p = 13 - 13 + 2 \cdot 3 = 6$.

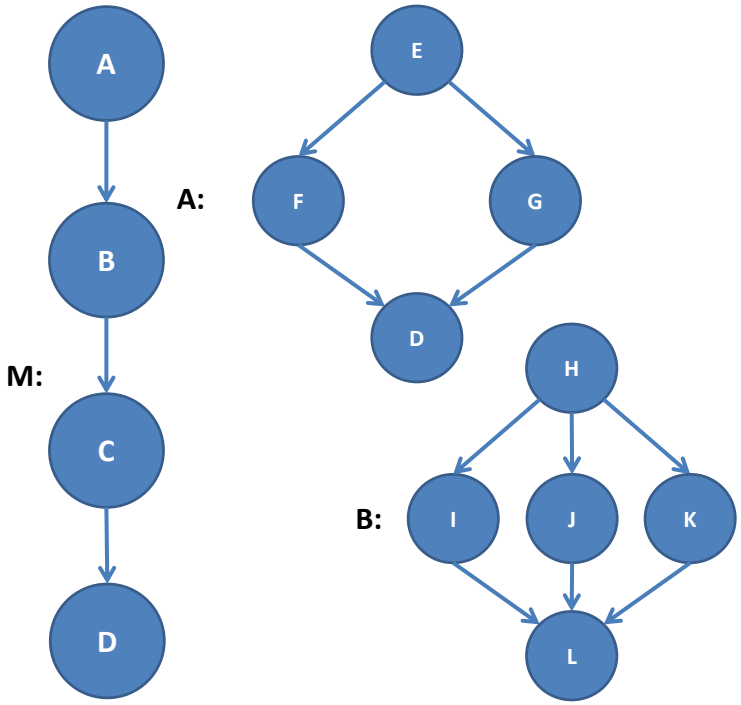

Figura 3.15: Grafo de fluxo de controle com mais de um componente.

Na abordagem proposta por McCabe, o número ciclomático do grafo de fluxo de controle de um programa não é apenas uma métrica de medida de complexidade. Ele tem papel importante nas atividades de teste caixa branca do programa indicando o número mínimo de caminhos a serem testados para uma cobertura completa. Para maiores informações sobre seu uso para finalidade de teste consulte (MCCABE, 1976), SOMMERVILLE (2006).

No artigo original de McCabe não existem limites rígidos definidos para a complexidade ciclomática de um programa. No entanto, nos experimentos apresentados no texto original o autor sugeriu a programadores que desenvolvessem módulos com complexidade ciclomática menor ou igual a 10 e no relato do mesmo experimento ele comenta que módulos com CC entre 3 e 7 estão muito bem estruturados e tendem a facilitar a tarefa de manutenção do código gerado (MCCABE, 1976).

3.2.10 Métricas de Herança

O conceito de encapsular atributos e comportamento em um tipo de dado não é exclusivo da orientação a objetos, a própria programação por tipos abstratos de dados segue esse mesmo conceito. O que diferencia a orientação a objetos é o conceito de herança (LORENZ e KIDD, 1994).

Herança é o nome dado ao mecanismo que permite que características comuns a diversas classes sejam reunidas em uma classe base, também chamada de superclasse (SOMMERVILLE, 2006). A partir da superclasse, outras classes podem ser especificadas. Através da herança, uma classe derivada, ou subclasse, carrega as

características (atributos e métodos) da classe base e as novas características que a diferencia da superclasse e das outras subclasses.

A Figura 3.16 apresenta um exemplo simples de hierarquia de herança entre classes representada usando notações UML. Neste exemplo, a superclasse *Animal* possui sete subclasses ou classes descendentes (Herbívoro, Carnívoro, Onívoro, Coelho, Leão, Hiena e Homem) com as quais compartilha suas características sob a regência dos modificadores de acesso da linguagem. Esta estrutura pode ser vista como uma árvore, onde a classe Animal, que representa uma generalização, é a raiz da árvore e as subclasses Coelho, Leão, Hiena e Homem são as folhas da árvore e representam conceitos mais especializados, além de compartilhar as características herdadas das superclasses também podem usufruir de mecanismos como o polimorfismo (SOMMERVILLE, 2006).

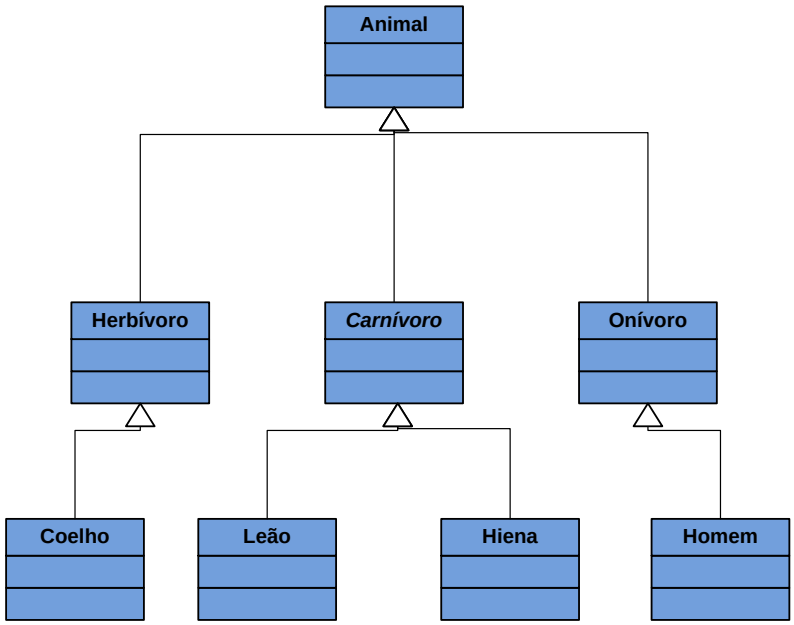

Figura 3.16: Exemplo de árvore de herança.

As métricas descritas a seguir foram propostas por Chidamber e Kemerer (CHIDAMBER e KEMERER, 1994) especificamente para avaliar o produto de software desenvolvido com base na orientação a objetos, quanto à aplicação do conceito de herança.

3.2.10.1 Profundidade da Árvore de Herança (DIT – do inglês Depth of Inheritance Tree)

A profundidade da árvore de herança mede o número de ancestrais de uma classe que estão no maior caminho de uma classe até a classe raiz da estrutura hierárquica de herança. De um ponto de vista semântico a métrica DIT permite ao projetista avaliar quantas superclasses podem afetar uma classe (CHIDAMBER e KEMERER, 1994).

Uma árvore de herança é uma representação da hierarquia de herança de um conjunto de classes em forma de uma árvore. A profundidade de um nó em uma árvore é o número de ramos a serem percorridos do nó até a raiz da árvore (CORMEN, LEISERSON, et al., 2001).

O exemplo apresentado na Figura 3.17 exibe uma árvore que representa uma hierarquia de herança. Nessa árvore a raiz é a classe Objeto, que é a superclasse da

classe Componente compartilha suas características com as classes abaixo dela na hierarquia. A classe raiz tem profundidade zero (DIT = 0), as classes no segundo nível da árvore têm profundidade um (DIT = 1), no exemplo da figura a classe Componente, as classes do terceiro nível tem profundidade dois (DIT = 2), como a classe Controle, e assim por diante.

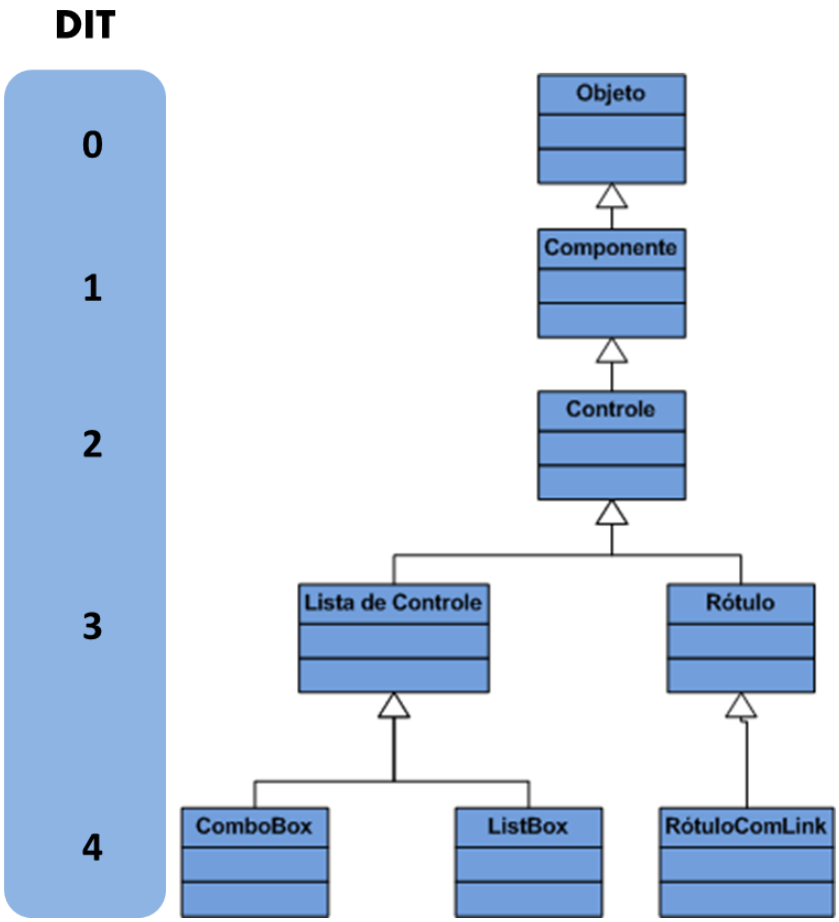

Figura 3.17: Exemplo de Profundidade da árvore de herança.

Os valores medidos para a DIT devem pertencer à classe dos números naturais, visto que não existem profundidades negativas ou fracionárias em hierarquias de herança. Os valores esperados para a métrica dependem de decisões de projeto e por isso podem variar de projeto para projeto. No entanto, Chidamber e Kemerer sugerem que o impacto da utilização do mecanismo de herança seja avaliado por meio da métrica DIT, pois essa métrica pode ser usada para avaliar os seguintes fatores (CHIDAMBER e KEMERER, 1994):

- quanto maior a profundidade de uma classe em uma hierarquia de herança maior a probabilidade de que ela herde mais métodos, tornando mais complexa a tarefa de predição de seu comportamento;
- árvores de hierarquia mais profundas indicam projetos mais complexos, uma vez que existem mais classes e métodos no projeto;
- quanto maior a profundidade de uma classe na hierarquia de herança maior a probabilidade de que ela faça reuso dos métodos de suas superclasses.

Segundo Rosenberg (1998), valores de DTI entre dois e três indicam um grau adequado de reuso, no entanto o autor também ressalta que quando a maioria dos ramos da árvore de herança for pouco profunda (DTI < 2) há um indicio de má exploração do desenvolvimento OO. Por outro lado, Ambler (1998) defende que se uma árvore de herança tem ramos com profundidade maior que cinco pode ser necessário reavaliar o projeto, devido ao aumento de complexidade gerado pela aplicação do mecanismo de herança.

3.2.10.2 Número de Filhos (NoC – do inglês Number of Children)

O número de filhos de uma classe contida em uma hierarquia de herança é o número de subclasses que são descendentes diretos dessa classe (CHIDAMBER e KEMERER, 1994).

Alguns fatores podem variar no cálculo da métrica de acordo com peculiaridades das linguagens como, por exemplo, na linguagem de programação Java onde uma classe que implemente uma interface é considerada uma filha direta dessa interface.

A Figura 3.18 apresenta uma árvore de hierarquia de herança simples, bem como marcações indicando o número de filhos e a profundidade da árvore de herança das classes representadas por quadrados. Nota-se que a raiz da árvore possui como descendentes diretos todas as outras classes na hierarquia o que é denotado pela indicação NOC = 7.

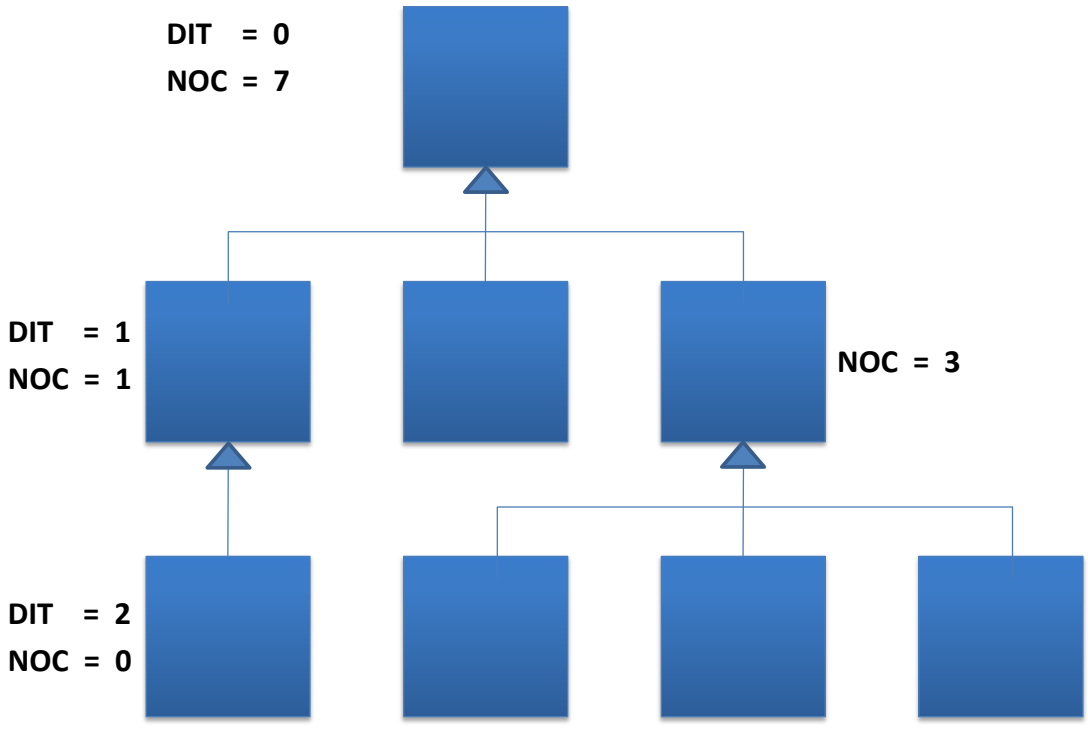

Figura 3.18: Exemplo de número de Filhos.

Os valores medidos para a métrica devem ser números naturais, já que as classes podem ter zero ou mais descendentes. Já os valores esperados para essa métrica podem variar de acordo com os requisitos do projeto e o papel que a classe a ser analisada representa dentro do projeto.

Segundo Hogan (1997), quanto maior o número de filhos maior o grau de reúso do código, visto que o mecanismo de herança é uma forma de reúso. Além disso, quanto

maior o número de descendentes de uma classe maior a probabilidade do projetista ter cometido erros no uso do mecanismo de herança, utilizando um nível inadequado de abstração para a superclasse da hierarquia (CHIDAMBER e KEMERER, 1994).

O número de filhos também pode indicar a influência de uma determinada classe no projeto como um todo, assim os métodos dessa classe podem exigir mais atenção na fase de testes (HOGAN, 1997)

3.2.10.3 Número de Métodos Redefinidos (NORM – do inglês Number of Overridden Methods)

Em um projeto orientado a objetos no qual o mecanismo de herança é aplicado, o número de métodos redefinidos (ou número de operações redefinidas) é a quantidade de métodos herdados das classes ancestrais que foram sobrescritos na subclasse em análise (LORENZ e KIDD, 1994).

No exemplo da Figura 3.19 apresentamos uma hierarquia de herança onde há redefinição de métodos. Na figura as classes Automóvel e Bicicleta redefinem os três métodos definidos na classe raiz (*checkList*, *adjust* e *cleanup*), e, portanto, apresentam *NORM=3*.

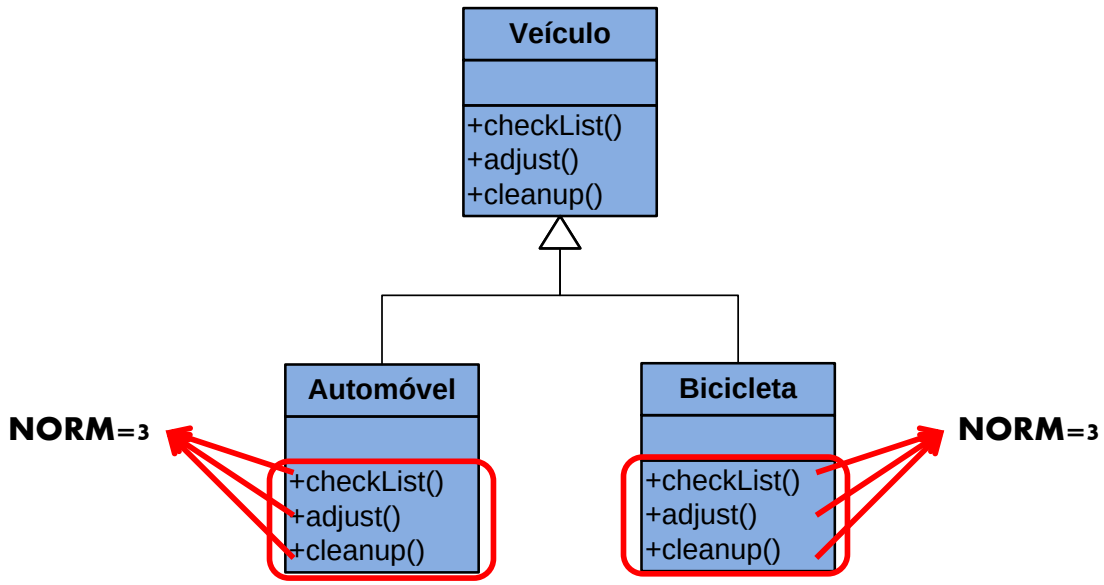

Figura 3.19: Exemplo de uma hierarquia de herança com redefinição de métodos.

Se uma classe tem um alto valor de *NORM* isso pode indicar um problema estrutural, visto que se muitas subclasses têm vários serviços redefinidos. Isso indica que provavelmente as subclasses não foram inseridas em uma hierarquia com superclasses adequadas. Além disso, segundo Lorenz e Kidd (1994), o número de métodos sobrescritos por uma subclasse deveria ser em torno de três. E este número deveria diminuir conforme a classe se aprofunda na hierarquia da árvore de herança.

3.2.11 Índice de Especialização (SI – do inglês *Specialization Index*)

Conforme Lorenz e Kidd (1994), a especialização em seu conceito puro implica na adição de mais funcionalidades a uma subclasse de forma a completar as funcionalidades já existentes nas classes ancestrais. Na prática, especialização de subclasses inclui outros fatores como:

a) a adição de novos métodos;

b) a extensão métodos existentes nas classes ancestrais, até mesmo chamando o método do ancestral dentro da nova implementação;

c) a redefinição de métodos com uma implementação totalmente nova;

d) a exclusão de métodos herdados.

Através da associação de algumas métricas pode-se avaliar a qualidade das subclasses quanto à aplicação do mecanismo de herança. Nesse contexto, podem-se ter subclasses de dois tipos: especialização e implementação (LORENZ e KIDD, 1994).

Uma subclasse que implementa adequadamente um novo tipo, estendendo as funcionalidades de seus ancestrais, onde um baixo número de métodos é redefinido, poucos métodos são adicionados e nenhum, ou poucos, métodos são excluídos é considerada uma subclasse de melhor qualidade quanto a exploração do mecanismo de herança. Esse tipo de classe é chamado de subclasse por especialização.

Já uma subclasse que faz um uso conveniente de alguns métodos ou atributos das classes ancestrais, mas não estende sua funcionalidade é denominada subclasse por implementação. Esse tipo de subclasse é criado para aproveitar alguns recursos da superclasse e não por ser realmente um subtipo da superclasse.

Segundo Lorenz, uma medida da especialização de uma classe pode ser obtida pela equação:

$$SI = \frac{NORM \cdot DIT}{NOM} \qquad \text{Equação 3.4}$$

onde:

SI = Índice de Especialização da classe;

NORM = Número de métodos Redefinidos pela Classe;

DIT = Profundidade da Árvore de Herença;

NOM = Número total de métodos da classe.

Lorenz define que os valores medidos para a métrica índice de especialização, não devem ultrapassar 15%, este limiar foi denominado o limite de normalidade (LORENZ e KIDD, 1994). Isso sugere que classes em níveis mais profundos da árvore de hierarquia de herança devem ter cada vez menos métodos redefinidos, assim, maximizando o reúso dos conceitos definidos nos seus ancestrais. Em outras palavras, conforme aumenta a profundidade na árvore de herança as classes devem estender as funcionalidades de suas ancestrais e realizar cada vez menos modificações nas funcionalidades herdadas.

3.2.12 Métodos Ponderados por Classe (WMC – do inglês *Weighted Methods per Class*)

Segundo Rosemberg (1998), a métrica métodos ponderados por classe mede a complexidade individual de uma classe. O número de métodos de uma classe, bem como suas complexidades, é um indicador do tempo e esforço para o desenvolvimento e a manutenção da classe. Por isso, para uma dada classe essa métrica é representada como o somatório das complexidades dos métodos da classe; como complexidade dos métodos geralmente assume-se a sua medida de complexidade ciclomática. A seguir apresentamos a fórmula da métrica WMC proposta por Chidamber e Kemerer (1994):

$$WMC = \sum_{i=1}^{n} C_i$$ Equação 3.5

onde *n* é o número de métodos da classe e *C* é a complexidade ciclomática dos métodos da classe.

De acordo com Chidamber e Kemerer (1994), classes com muitos métodos tendem a causar grande impacto em suas subclasses, uma vez que seus descendentes herdam seus métodos. Além disso, classes com métodos muito complexos tendem a ser muito especializadas, dificultando, assim, seu reuso em projetos futuros (CHIDAMBER e KEMERER, 1994). Por esses motivos, as classes devem apresentar valores para a métrica WMC tão baixos quanto forem possíveis (PRESSMAN, 2002).

3.2.13 Falta de Coesão dos Métodos (*LCOM – do inglês Lack of Cohesion of Methods*)

Segundo Pressman (2002), coesão é uma indicação qualitativa do grau em que um módulo se concentra em apenas uma coisa. No contexto da orientação a objetos dizemos que um método é coeso se ele é responsável por um conjunto reduzido de tarefas, no limite apenas uma (MARTIN, 1995).

Para Rosenberg (1998), a falta, ou ausência, de coesão é definida pelo número de diferentes métodos de uma classe que referenciam uma determinada variável de instância. Para Chidamber e Kemerer (1994), um módulo altamente coeso deveria manter sua funcionalidade mesmo sozinho. Dessa forma, uma boa divisão das classes além de propiciar a simplicidade dos módulos e auxiliar seu reuso também indica uma alta coesão. Métodos pouco coesos aumentam a complexidade do produto de software, o que pode aumentar os erros durante o processo de desenvolvimento.

Na visão de Chidamber e Kemerer (1994), a coesão dos métodos de uma classe deve ser avaliada, pois propicia o encapsulamento. Existem diversos trabalhos que propõem métricas para avaliar a coesão dos métodos de classes, dentre eles se destacam Rosenberg (1998), Chidamber e Kemerer (1994), Hitz e Montazeri (1996), Henderson-Sellers (1996), etc.

Em nossos experimentos utilizaremos a métrica proposta por Henderson-Sellers (1996), onde a falta de coesão dos métodos de uma classe é definida como:

$$LCOM = \frac{\left(\frac{1}{a}\cdot\sum_{j=1}^{a}\mu(A_j)\right)-m}{1-m}$$ Equação 3.6

onde:

m é o número de métodos;

a é o número de atributos;

A é o conjunto dos atributos da classe;

$\mu(A_j)$ é uma função que retorna o número de métodos da classe que acessa cada atributo, onde *j = 1 … a.*

Segundo Chidamber e Kemerer (1994) e Rosenberg (1998) as classes que apresentam baixos valores para métricas de avaliação da coesão devem ser divididas em classes menores e mais coesas.

3.2.14 Métricas de Acoplamento

Acoplamento é a medida de interconexão entre módulos em uma estrutura de software. O acoplamento é dependente da complexidade das interfaces entre os módulos, do ponto de entrada ou referência a um módulo e de que dados passam através da interface. Em outras palavras, o acoplamento é uma indicação quantitativa do grau em que um módulo está conectado a outros módulos e ao mundo externo (PRESSMAN, 2002).

No projeto de software o acoplamento deve ser mantido tão baixo quanto for possível. Manter as conexões entre os módulos simples resulta em um produto de software mais fácil de entender e manter, além de ser menos propenso a propagação de erros através do sistema (PRESSMAN, 2002).

Segundo Chidamber e Kemerer (1994), o acoplamento excessivo entre classes de objetos prejudica o projeto modular e impede o reúso, pois quanto mais independente for a classe mais fácil será de reutilizá-la em outro projeto. Os referidos autores também ressaltam que, para melhorar a modularidade e promover o encapsulamento, o acoplamento entre classes deve ser evitado tanto quanto possível, uma vez que esses relacionamentos entre as diferentes classes do projeto dificultam a manutenção do código. Nesse contexto, os autores sugeriram uma métrica específica para a avaliação de acoplamento com o intuito de direcionar maiores esforços às atividades de teste de classes mais acopladas. Essa métrica foi nomeada acoplamento entre objetos e será apresentada nesta seção.

A Figura 3.20 apresenta um exemplo de sistema acoplado, onde um pacote agrupa varias classes e esses módulos se relacionam entre si. As setas dirigidas indicam os relacionamentos entre as entidades e indicam a direção da dependência, ou seja, uma seta que parte de uma entidade A em direção a uma entidade B indica uma dependência da entidade B para com algum recurso da entidade A.

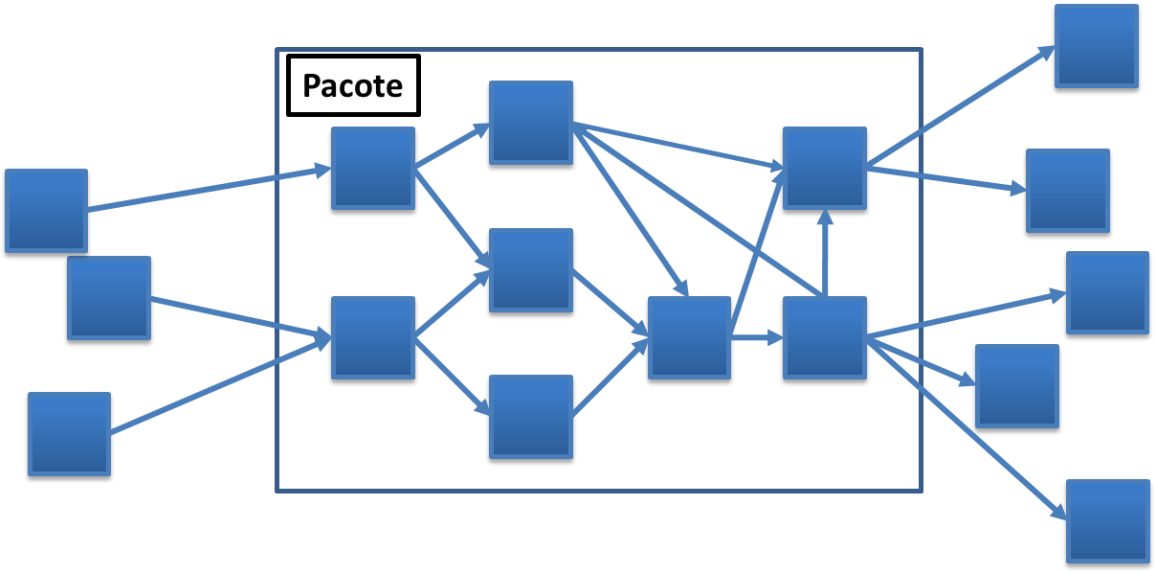

Figura 3.20: Exemplo de um sistema acoplado.

Diferentemente de Chidamber e Kemerer, Robert Martin propõe que o acoplamento seja analisado de forma diferenciada de acordo com a direção do relacionamento de dependência, a Figura 3.21 exibe essa distinção dos tipos de acoplamento (MARTIN,

1995), aferente e eferente; nas seções que seguem essas duas métricas serão apresentadas.

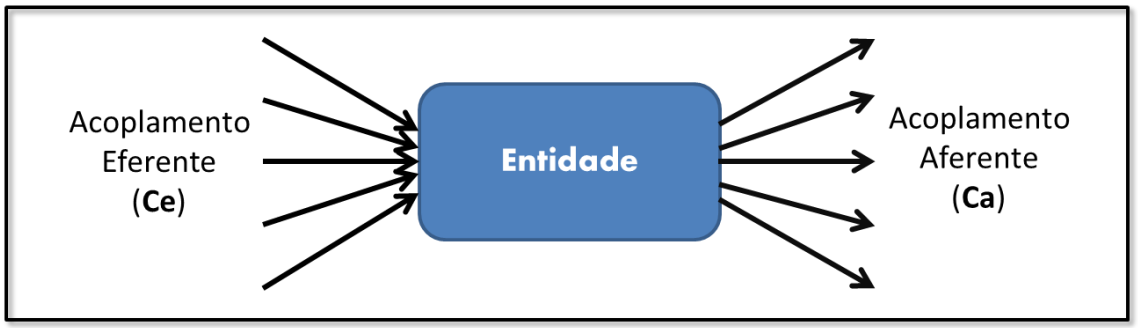

Figura 3.21: Distinção entre os tipos de acoplamento.

3.2.14.1 Acoplamento Aferente (Ca – do inglês Afferent Coupling)

A métrica Acoplamento Aferente proposta por Martin (1995) é dada pelo número de entidades que dependem da entidade em análise. Por entidade entenda-se qualquer unidade modularizada, como classes, pacotes, etc.

No exemplo da Figura 3.20 o acoplamento aferente do pacote é cinco, pois existem cinco classes fora do pacote que dependem de classes dentro do pacote.

3.2.14.2 Acoplamento Eferente (Ce – do inglês Efferent Coupling)

Contrapondo-se ao Acoplamento Aferente Martin propôs a métrica de acoplamento eferente com o objetivo de avaliar o quão dependente uma entidade é de outras entidades do projeto (MARTIN, 1995).

O exemplo da Figura 3.20 apresenta um acoplamento eferente de dois, haja vista que existem duas classes dentro do pacote que dependem de classes fora do pacote.

3.2.14.3 Acoplamento entre Objetos (CBO – do inglês Coupling Between Object Classes)

Para avaliar o acoplamento entre classes Chidamber e Kemerer (1994) propuseram a métrica CBO. Essa medida compartilha informações de ambas às métricas propostas por Martin (Ca e Ce). Segundo os autores o acoplamento pode ser medido como o número de todos os relacionamentos da classe em análise com outras classes, tanto os aferentes quanto os eferentes. Na Figura 3.20 o pacote exibido apresenta o valor 8 para a métrica CBO, visto que existem 8 relacionamentos entre classes do pacote e classes externas ao pacote.

Segundo Rosenberg (1998), o valor dessa métrica deveria ser menor ou igual a cinco, uma vez que classes com forte acoplamento podem ser mais difíceis de entender, manter e reusar.

3.2.15 Instabilidade e Abstração

3.2.15.1 Abstração (A – do inglês Abstractness)

Segundo Pressman (2002), a abstração é uma importante ferramenta para lidar com sistemas complexos. O referido autor atesta que em soluções modulares vários níveis de abstração podem ser propostos com o intuito de facilitar o entendimento e manuseio das informações envolvidas. Assim, abstração pode ser vista como um mecanismo que permite que alguém lide com um problema com algum grau de generalização sem

considerar detalhes de baixo nível (PRESSMAN, 2002). Em outras palavras, uma entidade é abstrata quando apresenta a qualidade de ser considerada a parte de uma instância específica, ou ainda quando é uma formalização mais genérica do conceito.

Segundo Martin (1995) a abstração de um determinado conjunto de entidades pode ser determinada pela seguinte equação:

$$A = \frac{(NCA+NI)}{NC}$$ Equação 3.7

onde:

NCA = Número de Classes Abstratas;

NI = Número de Interfaces;

NC = Número Total de Classes.

Essa métrica admite valores entre zero e um. Uma medida de abstração (*A*) próxima de zero (*0*) indica que as entidades analisadas são bastante concretas e que o mecanismo de abstração foi pouco utilizado, o que reduzirá a probabilidade de reúso da entidade. Já em escopos onde essa métrica se aproxima de um (*1*) temos entidades bastante abstratas e para as quais provavelmente não existem funcionalidades completamente implementadas (MARTIN, 1995). Como será discutido a seguir, o uso do mecanismo de abstração deve ser equilibrado de forma que sejam concebidas entidades que possam ser reusadas no futuro juntamente com entidades que implementam as funcionalidades necessárias ao projeto atual.

3.2.15.2 Instabilidade (I – do inglês Instability)

Martin propôs a métrica Instabilidade (*I*) para avaliar entidades quanto a sua dependência perante outras entidades (MARTIN, 1995). A fórmula proposta por Martin para avaliar a instabilidade de entidades é:

$$I = \frac{Ce}{(Ca+Ce)}$$ Equação 3.8

onde:

Ce é o acoplamento eferente da entidade;

Ca é o acoplamento aferente da entidade.

A métrica Instabilidade admite valores entre zero e um, onde uma entidade que tenha relações de dependência com várias outras é considerada instável. Já as entidades que dependem de um número pequeno de outras entidades são consideradas estáveis (MARTIN, 1995).

3.2.15.3 Instabilidade vs. Abstração

Segundo Martin, em um projeto nem todas as entidades podem ser totalmente estáveis (*I=0*), pois senão o sistema seria imutável. Ele afirma, também, que as entidades abstratas não podem ser instáveis, pois elas possibilitam a extensão de suas funcionalidades sem alterarem sua estabilidade. Isso pode ser feito através da declaração de subclasses usando o mecanismo de herança. Assim, o autor propõe que as entidades totalmente estáveis devem ser as mais abstratas, e que ao ponto que as entidades vão descendo na árvore de herança (se tornando mais concretas) e se associando a outras entidades, elas se tornam mais instáveis (MARTIN, 1995).

Um sistema não pode ser composto apenas de componentes abstratos, por outro lado deseja-se a possibilidade de expandir as funcionalidades dos sistemas, portanto, é preciso durante o projeto balancear a abstração e a instabilidade das entidades que compõem o sistema (MARTIN, 1995).

Para Martin, quando as entidades conseguem manter o compromisso do balanceamento da instabilidade e da abstração elas podem ser consideradas melhores que as entidades incapazes de manter esse compromisso. Para formalizar esse balanceamento o autor propôs que um plano cartesiano fosse gerado tendo como eixo das abscissas a métrica de Instabilidade (*I*) e como eixo das ordenadas as medidas de Abstração (*A*). Desta forma, pode-se traçar uma linha partindo das coordenadas de uma entidade completamente abstrata (*A=1*, *I=0*) até uma classe completamente concreta (*A=0*, *I=1*), como ilustrado n o gráfico da Figura 3.22.

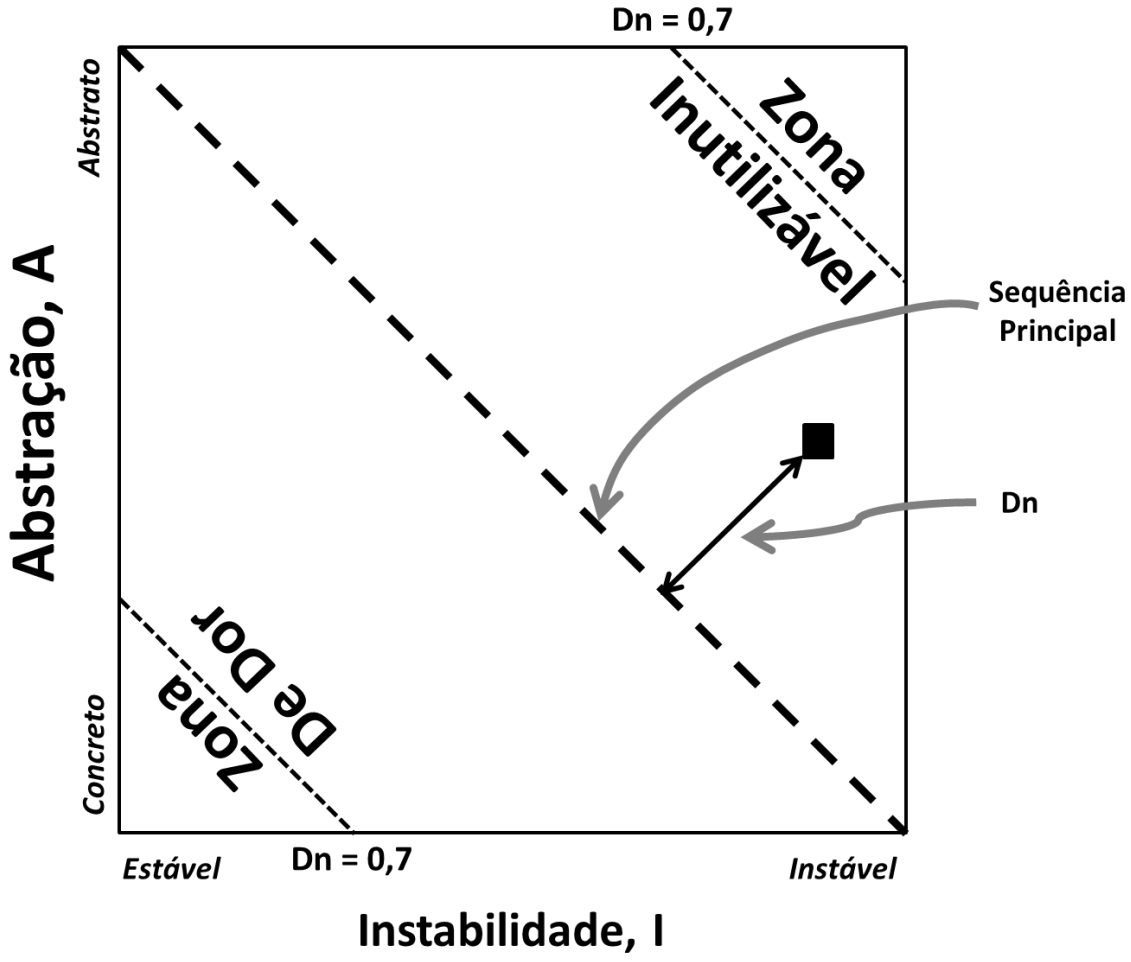

Figura 3.22: Gráfico de Abstração *vs*. Instabilidade.

Por este gráfico ser semelhante a um gráfico bastante difundido na Astronomia Martin, nomeou-o da mesma forma: Sequência Principal. Para Martin, as entidades que se situam sobre, ou próximas, a sequência principal não são nem muito abstratas para seu nível de estabilidade, nem muito instáveis para seu nível de abstração (MARTIN, 1995).

3.2.15.4 Distância Normalizada da Sequência Principal (Dn – do inglês Normalized Distance from Main Sequence)

Segundo Martin, quando as métricas de Abstração e Instabilidade de entidades se encontram próximas, ou mesmo sobre, a linha da sequência principal tem-se entidades onde abstração e instabilidade estão balanceadas (MARTIN, 1995). Então, para medir o balanceamento entre abstração e estabilidade Martin propôs a métrica Distância da Sequência Principal como sendo a distância euclidiana entre a linha que define a sequência principal e o ponto que define a entidade em termos das métricas de Abstração e Instabilidade. Essa métrica pode ser obtida pela fórmula a seguir:

$$Dn = |A + I - 1|$$ Equação 3.9

onde:

A é a medida de abstração da entidade;

I é a medida de instabilidade da entidade.

Essa métrica admite valores entre zero e um, onde valores próximos de zero (*0*) indicam entidades onde o compromisso de balancear a estabilidade e a abstração foi mantido e entidades com *Dn* próximo de um (*1*) indicam que essa entidade é ou muito abstrata para o seu nível de estabilidade ou muito instável pro seu nível de abstração (MARTIN, 1995).

4 DA PREDIÇÃO DE CARACTERÍSTICAS FÍSICAS

Dada à importância das características físicas para os sistemas embarcados, cada vez faz-se mais necessária à criação de alternativas que acelerem o processo de sua obtenção. Os processos convencionais de simulação em baixos níveis de abstração não são compatíveis com as necessidades de *time-to-market* que um produto precisa obedecer nos dias atuais, e, por isso, métodos preditivos ganham espaço e importância nos atuais complexos processos de desenvolvimento.

Neste capítulo apresentaremos os experimentos realizados para estimar os consumos de características físicas. Esses experimentos têm o intuito de explorar novas formas de estimar o consumo de recursos de hardware de uma forma mais rápida, possibilitando, assim, que desobediências às restrições físicas de projeto de determinadas implementações sejam percebidas mais rapidamente, então aumentando a eficiência do processo de desenvolvimento.

Em todos os experimentos apresentados neste capítulo foram utilizadas aplicações implementadas em linguagem de programação voltada ao paradigma orientado a objetos, mais especificamente a linguagem Java. Além disso, as aplicações foram compiladas para a plataforma alvo, que neste trabalho é o processador FemtoJava multiciclo. Ele foi escolhido porque sua dissipação de potência é mais adequada a sistemas embarcados.

As métricas de qualidade de software, apresentadas no Capítulo3, foram obtidas a partir do código fonte das aplicações através do uso da ferramenta *Eclipse Metrics plug-in* (METRICS, 2005). Por sua vez, as métricas físicas, ou métricas de baixo nível, foram extraídas do trabalho de Redin (2008), onde o desempenho (ciclos) obtido pelo uso do simulador com precisão de ciclo – CACO-PS (BECK e CARRO, 2003) – e para consumo de energia através da ferramenta *Synopsis Power Compiler* (SYNOPSYS, 2010), a partir da síntese das descrições VHDL do processador. Além disso, as métricas relativas a ocupação de memória, tanto de dados quanto de programa, foram obtidas a partir da ferramenta de estimativas DESEJOS (MATTOS e CARRO, 2007).

Para a predição das características físicas a partir das métricas de qualidade de software foram utilizadas Redes Neurais Artificiais (RNA). Um modelo de rede *Multi Layer Perceptron* (MLP) foi adotado dentre os diversos tipos e topologias de RNAs. Esse é o tipo mais comum de redes neurais (HAYKIN, 1994), além disso, este tipo de rede, organizado em uma topologia de duas camadas com processamento ativo, é capaz de ser utilizado como um método aproximador universal de funções, como demonstrado por Cybenko (1989).

Para simplificar a escrita do texto, a lista completa dos parâmetros de treinamento das redes neurais utilizadas nos experimentos apresentados nesse capítulo é apresentada no Apêndice B.

4.1 Experimentos

Uma vez coletadas as métricas de qualidade de software e as métricas físicas, os dados foram, então, utilizados no treinamento e teste de Redes Neurais Artificiais (RNA) e de um mecanismo de regressão estatística multivariada, assim obtendo-se modelos preditivos da variação de consumo de recursos físicos baseados nas diferentes implementações da aplicação.

4.1.1 Experimentos iniciais

Nesta etapa de nossos experimentos diferentes implementações de duas aplicações foram caracterizadas, em termos de métricas de qualidade de software e de métricas físicas. As aplicações utilizadas foram: um decodificador Mpeg Audio Layer-3 e o algoritmo de controle de uma cadeira de rodas.

Tabela 4.1: Métricas Físicas das Três Implementações do MP3.

	Implem. 1	*Implem. 2*	*Implem. 3*
Memória de Dados (bytes)	238.484	237.192	242.688
Memória de Programa (bytes)	146.812	117.756	324.733
Ciclos	1.830.675.876	830.365.894	239.748.559
Energia (J)	79,8575	36,2221	21,9624

Tabela 4.2: Métricas de Qualidade de Software das Três Implementações do MP3.

	Implem. 1	*Implem. 2*	*Implem. 3*
Abstração	0,143	0	0,2
Acoplamento Aferente	3	2	5
Acoplamento Eferente	4	2	3
Complexidade Ciclomática	1,832	7,448	6,492
Distância Normalizada	0,707	0,556	0,626

Falta de Coesão dos Métodos	0,655	0,42	0,245
Instabilidade	0,75	1	1
Métodos Ponderados por Classe	1.081	648	766
Número de Atributos	186	112	6
Número de Atributos Estáticos	98	58	597
Número de Classes	27	26	64
Número de Filhos	106	22	4
Número de linhas de Código dos Métodos	4.101	4.618	5.675
Número de Métodos	463	85	47
Número de Métodos Estáticos	127	2	71
Número de Pacotes	5	3	6
Número de Parâmetros	7	14	6
Número Total de Linhas de Código	7.891	6.853	8.423
Profundidade da Árvore de Herança	2	1	2
Profundidade de Blocos Aninhados	1,188	2,23	2,305

A aplicação Mpeg Audio Layer-3 tem três diferentes implementações, cujos consumos de energia e ciclos de cada uma das três implementações são apresentados na Tabela 4.1, bem como suas métricas de qualidade de software são apresentadas na Tabela 4.2.

Tabela 4.3: Métricas Físicas das Quatro Implementações do Controle da Cadeira de Rodas.

	Implem. 1	*Implem. 2*	*Implem. 3*	*Implem. 4*
Memória de Dados (bytes)	2.063	6.248	5.208	5.094
Memória de Programa (bytes)	372	582	431	421
Ciclos	1.898	28.588	9.104	7.776
Energia (J)	2.714.132	40.569.570	12.916.022	11.026.748

Tabela 4.4: Métricas de Qualidade de Software das Quatro Implementações do Controle da Cadeira de Rodas.

	Implem. 1	*Implem. 2*	*Implem. 3*	*Implem. 4*
Abstração	0	0	0	0
Acoplamento Aferente	1	4	2	2
Acoplamento Eferente	1	2	2	2
Complexidade Ciclomática	1,238	1,312	1,261	1,25
Distância Normalizada	0,5	0,567	0,583	0,583
Falta de Coesão dos Métodos	0,71	0,639	0,51	0,519
Instabilidade	1	0,5	0,667	0,667
Métodos Ponderados por Classe	26	42	29	25
Número de Atributos	0	22	20	17
Número de Atributos Estáticos	20	28	17	19

Número de Classes	5	7	7	7
Número de Filhos	0	0	0	0
Número de linhas de Código dos Métodos	58	94	62	49
Número de Métodos	2	29	20	17
Número de Métodos Estáticos	8	3	3	3
Número de Pacotes	3	4	4	4
Número de Parâmetros	3	3	3	2
Número Total de Linhas de Código	146	283	190	170
Profundidade da Árvore de Herança	1	2	2	2
Profundidade de Blocos Aninhados	1,143	1,25	1,174	1,15

Para a aplicação de controle de uma cadeira de rodas foram avaliadas quatro implementações diferentes. As métricas de baixo nível para cada uma dessas quatro implementações são apresentadas na Tabela 4.3, já as métricas de alto nível para as mesmas são apresentadas na Tabela 4.4.

No primeiro experimento realizado foram utilizados os dados de caracterização das duas aplicações, MP3 e controle da cadeira de rodas, para o treinamento de uma rede neural artificial. Esses dados foram associados em um único conjunto composto por 7 amostras a serem apresentadas a uma rede neural artificial. Essas amostras consistem das métricas (de alto e baixo nível) de 3 versões do MP3 e 4 do controle da cadeira de rodas. Neste experimento os dados foram separados em 7 conjuntos de treinamento (compostos por 6 dos 7 exemplos do conjunto original) e 7 conjuntos de teste (composto por uma amostra do conjunto original). Foram treinadas sete RNAs, uma para cada subconjunto do conjunto de dados original.

Tabela 4.5: Erros Percentuais de uma RNA para a Predição das Métricas de Baixo Nível do MP3 e do algoritmo de controle da cadeira de rodas.

	[2-7]	*[1] [3-7]*	*[1 - 2][4-7]*	*[1 - 3][5-7]*	*[1 - 4][6-7]*	*[1 - 5][7]*	*[1 - 6]*
Memória de	2,3341	13,154	2,0094	0,06384	1088,6	135,04	78,208

Dados (bytes)							
Memória de Programa (bytes)	10,016	1,4206	0,19161	0,13242	4,1426	116,81	108,53
Ciclos	32,286	8,3478	2,7904	0,19431	148,76	595,07	338,53
Energia	780,18	27,684	787,63	138,49	83,283	217,98	211,61

Foi utilizada uma rede neural artificial do tipo Perceptron Multicamadas (MLP – do inglês *Multi Layer Perceptron*), propagada para frente, com algoritmo de treinamento de Levenberg-Marquardt (HAYKIN, 1994). Essa RNA possuía 23 neurônios sensoriais na camada de entrada, 20 neurônios, com função de ativação tangente hiperbólica na camada escondida e, por fim, 4 neurônios com função de ativação linear na camada de saída, um para cada métrica física.

A Tabela 4.5 apresenta, em cada coluna, o erro percentual de teste para um exemplo não apresentado durante a etapa de treinamento. O rótulo da coluna indica qual das amostras foi utilizada como exemplo de teste, todas as outras amostras foram utilizadas no processo de treinamento da rede neural artificial.

Os resultados da metodologia original de predição apresentados na Tabela 4.5 mostraram-se bastante ineficientes, demonstrando que a RNA não fora capaz de convergir para uma boa aproximação dos dados apresentados. Como podemos ver na referida tabela, em alguns casos obtivemos erros de quase 800%, o que indica que a metodologia deve ser revista para facilitar o aprendizado da rede neural, bem como a posterior predição.

A primeira abordagem adotada para a simplificação dos padrões de entrada da rede foi a separação das diferentes aplicações em conjuntos distintos de dados. Assim, esperava-se retirar dos dados de treinamento variações relativas às entradas das aplicações, uma vez que a rede não recebe nenhuma informação sobre elas e a execução das implementações é altamente dependente das mesmas. Essa modificação é apresentada na seção a seguir.

4.1.2 Experimentos de predição para uma única aplicação

Os experimentos de predição das características físicas do MP3 foram realizados da seguinte forma: todas as métricas das três implementações foram agrupadas em um conjunto de treinamento que foi apresentado a uma RNA do tipo MLP alimentada pra frente, como já fora dito anteriormente.

Essa rede neural é constituída por três camadas, onde uma é camada de entrada com 23 neurônios, um para cada métrica de alto nível, outra é a camada escondida, composta por 3 neurônios com função de ativação tangente hiperbólica, e, por fim, uma camada de saída com 4 neurônios, um para cada métrica de baixo nível. Devido ao baixo número de exemplos neste experimento os conjuntos de treinamento e de teste da rede neural foram os mesmos.

No contexto apresentado, a baixa magnitude, tendendo a zero, dos erros apresentados na Tabela 4.6 já seria esperada, visto que a rede teve condições de minimizar os erros para estes dados durante a etapa de treinamento.

Tabela 4.6: Erros Absolutos de uma RNA para a Predição das Métricas de Baixo Nível do MP3.

	Implem. 1	*Implem. 2*	*Implem. 3*
Memória de Dados (Bytes)	-5,014E-14	-4,7E-14	0
Memória de Programa (Bytes)	-4,93E-14	-4,2E-14	3,95E-14
Ciclos	2,0674E-14	-1,5E-13	-4,9E-14
Energia (J)	1,979E-14	-1,7E-14	4,2E-14

O experimento de predição das características físicas do controle da cadeira de rodas foi conduzido de forma um pouco diferente do experimento anterior. Nesse experimento, os dados, de alto e baixo nível, da aplicação foram separados gerando diferentes conjuntos de treinamento e de teste. Os conjuntos de treinamento são subconjuntos do conjunto contendo todos os exemplos. Cada um destes subconjuntos é constituído pelas métricas de 3 das 4 implementações, 3 amostras das 4 existentes. O exemplo restante não é apresentado a RNA na etapa de treinamento, constituindo, assim, o conjunto de teste. Os valores apresentados na Tabela 4.7 são os erros, relativos e percentuais, de predição para aqueles exemplos que não foram apresentados durante o treinamento da rede, desta forma podemos explorar a capacidade de predição da RNA para exemplos não apresentados durante o treinamento.

Tabela 4.7: Erros Percentuais de uma RNA para a Predição das Métricas de Baixo Nível do Controle da Cadeira de Rodas.

	Implem. 1	*Implem. 2*	*Implem. 3*	*Implem. 4*
Memória de Dados (Bytes)	86,75	6,21	46,58	9,24
Memória de Programa (Bytes)	38,18	9,43	43,96	1,74
Ciclos	79,83	12,43	105,54	29,02
Energia (J)	70,16	0,29	71,68	23,21

A RNA utilizada nesse experimento é do tipo MLP, alimentada pra frente e usa o método da retropropagação dos erros para atualização dos pesos e o método de aprendizado proposto por Levemberg-Marquadt (HAYKIN, 1994). Além disso, essa rede é composta por 23 neurônios transparentes de entrada, um para cada métrica de qualidade de software, 5 neurônios na camada escondida, todos utilizando função de ativação tangente hiperbólica, e, por fim, 4 neurônios com função de ativação linear na camada de saída, um para cada métrica de baixo nível.

Analisando os resultados apresentados na Tabela 4.6 e na Tabela 4.7 nota-se que a rede neural não fora capaz de convergir para um estado de baixos erros de predição, mesmo após separarmos as métricas das duas aplicações em redes diferentes. No

próximo capítulo uma proposta de pré-processamento das métricas de alto nível será apresentada, este pré-processamento dos valores apresentados à RNA tem o intuito de melhor representar as aplicações executadas.

5 EXPERIMENTOS COM MÉTRICAS ESTRITAS E BAIXA VARIABILIDADE DOS DADOS

Os resultados dos experimentos apresentados no capítulo anterior demonstraram a necessidade da aplicação de técnicas de simplificação dos padrões de entrada. Então, neste capítulo, a possibilidade de pré-processar as métricas de qualidade de software de uma implementação é avaliada. Este pré-processamento faz-se necessário devido ao fato de as ferramentas de extração de métricas de alto nível a partir de código fonte, geralmente, incorporarem como medidas de uma métrica para uma implementação todo o código contido no projeto, mesmo código de teste ou código não utilizado de bibliotecas que estão associadas ao projeto em questão.

Para a realização deste pré-processamento foi utilizado um perfil de execução da aplicação para definir quais as classes existentes no projeto estavam realmente sendo utilizadas na aplicação, e, com base nessa informação, a saída da ferramenta de extração de métricas fora modificada de forma que somente as classes presentes no perfil de execução tivessem suas métricas contabilizadas. A esse processo excetuaram-se as métricas as quais a ferramenta realiza a extração por pacotes e não por classes (Abstração, Instabilidade, Acoplamentos e Distância Normalizada da Sequência Principal), isto porque a ferramenta impossibilita, em sua saída, que se possam selecionar apenas as métricas para os pacotes em uso. Para elas os valores de métricas utilizados foram os valores médios apresentados pela ferramenta *Eclipse Metrics Plugin* (METRICS, 2005).

Além do pré-processamento das métricas de alto nível nesta seção utilizamos uma mesma aplicação com um número maior de versões, ainda com o intuito de facilitar o treinamento da rede neural. A aplicação escolhida foi uma implementação de uma *Drink Vending Machine* (DVM) com diversas versões geradas automaticamente a partir de um modelo em *Alloy* (FERREIRA, 2008). Essas diferentes implementações consistem de 5 versões geradas automaticamente a partir de um mesmo modelo *Alloy* variando o tipo das duas principais estruturas de dados utilizadas na aplicação.

A seguir, na Tabela 5.1, são apresentadas as métricas de qualidade de software originalmente extraídas pela ferramenta *Eclipse Metrics Plugin* (METRICS, 2005) a partir das 5 versões da DVM. As métricas de código apresentadas representam o projeto inteiro, incluindo bibliotecas que implementam comportamentos da semântica da linguagem *Alloy* que não foram utilizados nesta aplicação.

Tabela 5.1: Métricas de qualidade de software originais das diferentes versões da DVM.

	alloyVM ListList	*alloyVM ListMap*	*alloyVM ListSet*	*alloyVM SetMap*	*alloyVM SetSet*
Abstração	0,314	0,314	0,314	0,314	0,314
Acoplamento Aferente	4,409	4,409	4,409	4,409	4,409
Acoplamento Eferente	5,636	5,636	5,636	5,636	5,636
Complexidade Ciclomática	803	812	840	842	701
Distância Normalizada da Sequência Principal	0,48	0,48	0,48	0,48	0,48
Falta de Coesão dos Métodos	12,084002	12,084002	11,906002	11,906002	9,092001
Índice de Especialização	7,733	7,733	8,304	8,304	8,048
Instabilidade	0,432	0,432	0,432	0,432	0,432
Métodos Ponderados por Classe	803	812	840	842	701
Número de Atributos	88	88	89	89	73
Número de Atributos Estáticos	50	50	52	52	47
Número de Classes	22	22	23	24	21
Número de Filhos	116	116	118	118	117
Número de Interfaces	1	0	0	0	0
Número de Linhas de Código dos Métodos	2248	2277	2353	2356	1951
Número de Métodos	412	414	431	433	361
Número de Métodos Estáticos	40	42	46	46	41
Número de Métodos Sobrecarregados	47	47	53	53	48
Número de Pacotes	7	7	7	7	7
Número de Parâmetros	301	307	322	323	264

Número Total de Linhas de Código	14683	14789	15241	15285	13047
Profundidade da Árvore de Herança	74	76	79	81	70
Profundidade de Blocos Aninhados	596	604	630	632	534

A Tabela 5.2, por sua vez, apresenta as medidas das métricas de qualidade de software, pré-processadas, das mesmas 5 versões da DVM, onde cada coluna representa uma das diferentes versões e cada linha representa uma das diferentes métricas de software extraídas.

Portanto, na Tabela 5.2 as métricas exibidas foram filtradas, de forma que as interferências das classes que não tenham sido efetivamente utilizadas na execução fossem descartadas. Para que esse processamento pudesse ser realizado fez-se necessário o uso de um perfil de execução da aplicação de forma a obter-se as classes executadas.

Comparando-se a Tabela 5.1 e a Tabela 5.2 pode-se notar uma variação expressiva nas métricas. Por exemplo, a métrica complexidade ciclomática tem uma variação de 701 para 161, na versão apresentada pela última coluna das tabelas. Outro bom exemplo é o número de linhas de código, onde também encontramos variações significativas, 15285 para 3895 linhas, na implementação da apresentada na penúltima coluna das tabelas.

Tabela 5.2: Métricas de qualidade de software processadas para as diferentes versões da DVM.

	alloyVM ListList	*alloyVM ListMap*	*alloyVM ListSet*	*alloyVM SetMap*	*alloyVM SetSet*
Abstração	0,314	0,314	0,314	0,314	0,314
Acoplamento Aferente	4,409	4,409	4,409	4,409	4,409
Acoplamento Eferente	5,636	5,636	5,636	5,636	5,636
Complexidade Ciclomática	151	160	163	165	161
Distância Normalizada da Sequência Principal	0,48	0,48	0,48	0,48	0,48
Falta de Coesão dos Métodos	4,569	4,569	4,391	4,391	4,391
Índice de Especialização	1,26	1,26	1,26	1,26	1,26
Instabilidade	0,432	0,432	0,432	0,432	0,432

Métodos Ponderados por Classe	151	160	163	165	161
Número de Atributos	22	22	22	22	22
Número de Atributos Estáticos	12	12	12	12	12
Número de Classes	16	17	17	18	16
Número de Filhos	95	95	95	95	95
Número de Interfaces	0	0	0	0	0
Número de Linhas de Código dos Métodos	447	476	496	499	494
Número de Métodos	80	82	78	80	76
Número de Métodos Estáticos	22	24	26	26	26
Número de Métodos Sobrecarregados	5	5	5	5	5
Número de Pacotes	6	6	6	6	6
Número de Parâmetros	46	52	54	55	53
Número Total de Linhas de Código	3727	3826	3858	3895	3823
Profundidade da Árvore de Herança	35	37	38	40	35
Profundidade de Blocos Aninhados	114	122	123	125	121

As métricas físicas, energia e ciclos, para cada uma das versões da aplicação DVM, são apresentadas na Tabela 5.3, onde cada linha representa uma das métricas de baixo nível e cada coluna representa uma das diferentes implementações da DVM.

Tabela 5.3: Métricas físicas das diferentes versões da DVM.

	alloyVM ListList	*alloyVM ListMap*	*alloyVM ListSet*	*alloyVM SetMap*	*alloyVM SetSet*
Ciclos	2134844775	2422809628	2537138897	2497437006	2383105150
Potência (mW)	23,22813774	23,26654085	23,22269468	24,16493251	24,25472074

Energia (J)	9,917693698	11,27407984	11,78384039	12,07007934	11,56030998

Para realizar a predição das características físicas desta aplicação a metodologia adotada será semelhante a dos experimentos do capítulo anterior. Os dados foram arranjados em conjuntos de treinamento, sempre deixando um dos exemplos, ou amostras, para servir de teste de predição.

Nesse experimento é utilizada uma rede neural artificial do tipo MLP, alimentada pra frente, com atualização dos pesos por retropropagação dos erros e com o método de aprendizado de Levemberg-Marquadt (HAYKIN, 1994). Esta RNA é composta por 3 camadas, uma camada de entrada com 23 neurônios sensoriais, ou transparentes, responsáveis por receber cada uma das métricas de software apresentadas na Tabela 5.2. A segunda camada, ou camada escondida, é composta por 20 neurônios com função de ativação tangente hiperbólica e, por fim, a última camada, ou camada de saída, é composta por um único neurônio, representando uma das métricas físicas. A Tabela 5.4 apresenta os erros percentuais de predição das RNAs, para exemplos não treinados, nas tarefas de predição de consumo de ciclos e energia.

Tabela 5.4: Erros percentuais de predição da RNA.

	alloyVM ListList	*alloyVM ListMap*	*alloyVM ListSet*	*alloyVM SetMap*	*alloyVM SetSet*
Ciclos	15,9219	5,88931	9,38821	3,41554	12,4414
Energia (J)	17,6411	0,55457	-4,88115	-7,76719	-2,59516

Baseado nos resultados de predição apresentados na Tabela 5.4, e na observação da baixa variabilidade das métricas de software na Tabela 5.2 propõe-se um novo experimento, onde novas versões de uma aplicação poderiam ser geradas automaticamente por alguma ferramenta, controlando-se as modificações no código para que as variações nas métricas de alto nível fossem pequenas, e dessa forma manter-se-iam os bons resultados de predição alcançados. De fato, existem trabalhos que abordam a capacidade das redes neurais extrapolarem os padrões aprendidos, desde que os padrões desconhecidos não extrapolem um determinado intervalo de confiança (YANG, KAVLI, et al., 2002).

Além disso, os dados da Tabela 5.2 indicam que dadas estas pequenas variações algumas métricas não variam de uma versão para outra (como número de pacotes e número de métodos sobrecarregados), e no âmbito das redes neurais características de entrada que não variam podem ser eliminadas visto que não contribuem na diferenciação dos exemplos (HAYKIN, 1994).

Essas abordagens de otimização da metodologia serão abordadas na seção que segue.

5.1.1 Abordagem de baixa variabilidade dos dados baseada em refatorações

Nos experimentos desta seção, utilizamos uma RNA alimentada pra frente do tipo MLP com três camadas para prever as características físicas de diferentes implementações de uma aplicação. A camada de entrada desta rede é composta por neurônios transparentes, ou sensoriais, um para cada métrica de software utilizada. Já a camada de saída possui um único neurônio, com função de ativação linear, que será

responsável pela predição de uma das métricas (ou características) físicas. Ainda, nestes experimentos as características físicas preditas serão ciclos e potência. Por fim, a camada intermediária, ou camada oculta, é composta por 25 neurônios, onde cada um deles possui conexões ponderadas com cada neurônio das outras duas camadas. Esses neurônios possuem função de ativação sigmóide, do tipo tangente hiperbólica. É essa característica que traz a capacidade de aprendizagem não linear à RNA (HAYKIN, 1994)

As seguintes aplicações foram utilizadas em nossos experimentos: decodificador mpegaudio (presente no conjunto de benchmarks para máquinas virtuais Java Spec JVM 2008 (SPEC, 2010)) e uma agenda de endereços simples. Ambas as aplicações sofreram sucessivas refatorações, do tipo método *inline* nos métodos que apresentavam maior consumo de ciclos na versão original da aplicação. Para cada aplicação, foi realizado o *inline* apenas dos métodos cujos consumos de ciclos, somados alcançavam ao menos 90% do consumo total de ciclos da aplicação.

As diferentes versões das aplicações consistem de versões refatoradas da implementação original. A Tabela 5.5 mostra os consumos de ciclos, potência (expresso em mili Watts) e energia (expresso em Joules) do decodificador Spec mpegaudio. As linhas da tabela mostram as diferentes versões, desde a original (mpegaudio) até a com o maior número de *inlines* incrementais (mpegaudioRef7).

Tabela 5.5: Métricas Físicas Extraídas do Decodificador mpegaudio, contido no pacote Spec JVM 2008.

	Ciclos	*Potência (mW)*	*Energia (J)*
mpegaudio	80147352885	13,82885354	221,669201
mpegaudioRef1	80107640093	13,82334563	221,471119
mpegaudioRef2	80057535324	13,88699787	222,351765
mpegaudioRef3	79290942036	13,82281201	219,204757
mpegaudioRef4	79338316236	13,97053844	221,679799
mpegaudioRef5	79041113315	14,20719847	224,590557
mpegaudioRef6	79039372865	14,22017402	224,790727
mpegaudioRef7	79037844665	14,22353079	224,839443

A Tabela 5.6 mostra as métricas de qualidade de software para as diferentes versões do decodificador Spec mpegaudio utilizadas nos experimentos. Nessa tabela podemos notar a redução incremental da métrica número de métodos, de 307 para 301, o que evidencia o emprego do *method inline* na forma de um método por iteração.

Tabela 5.6: Métricas de Software da Aplicação mpegaudio.

	Original	Ref1	Ref2	Ref3	Ref4	Ref5	Ref6	Ref7
Abstração	0,184	0,184	0,184	0,184	0,184	0,184	0,184	0,184
Acoplamento Aferente	19,7	19,7	19,7	19,7	19,7	19,7	19,7	19,7
Acoplamento Eferente	6,675	6,675	6,675	6,675	6,675	6,675	6,675	6,675
Complexidade Ciclomática	1316	1315	1314	1313	1313	1337	1336	1335
Distância Normalizada	0,338	0,338	0,338	0,338	0,338	0,338	0,338	0,338
Falta de Coesão nos Métodos	8,688	8,655	8,655	8,722	8,717999	8,717999	8,71	8,71
Instabilidade	0,519	0,519	0,519	0,519	0,519	0,519	0,519	0,519
Métodos Ponderados Por Classe	1316	1315	1314	1313	1313	1337	1336	1335
Número de Atributos	191	191	191	191	191	191	191	191
Número de Atributos Estáticos	141	141	141	141	141	141	141	141
Número de Classes	27	27	27	27	27	27	27	27
Número de Filhos	11	11	11	11	11	11	11	11

Número de Interfaces	0	0	0	0	0	0	0	0
Número de Linhas de Código dos Métodos	5236	5235	5236	5260	5262	5344	5342	5341
Número de Métodos Sobrescritos	6	6	6	6	6	6	6	6
Número de Métodos	307	306	305	304	303	303	302	301
Número de Métodos Estáticos	99	99	99	99	99	98	98	98
Número de Pacotes	5	5	5	5	5	5	5	5
Número de Parâmetros	409	409	408	408	407	401	398	396
Número Total de Linhas de Código	32350	32340	32330	32440	32435	32840	32815	32795
Profundidade da Árvore de Herança	50	50	50	50	50	50	50	50
Profundidade de Blocos Aninhados	641	640	639	638	636	635	631	627

Na Tabela 5.7 encontram-se os erros apresentados pela rede neural artificial e da regressão estatística multivariada na predição das características físicas apresentadas na Tabela 5.5. Tanto a rede neural, quanto a regressão foram treinados com os dados apresentados na Tabela 5.5 e na Tabela 5.6, utilizados como saída e entrada da RNA, respectivamente.

Nas etapas de treinamento, para ambos os métodos, o conjunto de treinamento foi construído baseado na metodologia *leave-one-out-cross-validation*. Ou seja, cada conjunto de treinamento era composto por todos os exemplos, exceto o exemplo que nomeia a sua respectiva coluna na Tabela 5.7. Assim sendo, na coluna intitulada "mpegaudio" são apresentados os erros de predição para um treinamento que incluía os exemplos refatorados (mpegaudioRef1 até mpegaudioRef7), portanto os erros da Tabela 5.7 são erros de exemplos não apresentados ao método de aprendizagem.

Tabela 5.7: Erros Percentuais de Predição Para Ciclos e Potência Para o Decodificador mpegaudio.

	Original	Ref1	Ref2	Ref3	Ref4	Ref5	Ref6	Ref7
RNA (Ciclos)	-0,55992	0,048621	-0,26091	0,058745	0,70905	-0,0036917	0,0033706	0,0030345
Regressão (Ciclos)	-0,42829	-0,046577	0,036963	0,033301	-0,028844	0,047205	-0,019922	0,032902
RNA (Potência)	-0,37912	0,25205	-0,67123	1,5111	-1,3157	-0,056251	0,22989	-3,233
Regressão (Potência)	-9,8265	-1,069	0,84451	0,76438	-0,65507	1,0542	-0,4445	0,73393

Devido à grande a magnitude dos dados apresentados na coluna de ciclos da Tabela 5.5 houve a necessidade de aplicação da normalização em magnitude para facilitar o aprendizado da rede neural. Para o cálculo dos erros (apresentados na Tabela 5.7) os dados de saída da rede foram desnormalizados e o cálculo do erro relativo percentual foi efetuado. Para a aplicação *AddressBook*, cujos experimentos são apresentados a seguir, esse pré-processamento dos dados não se fez necessário, visto que a rede foi capaz de lidar com os dados originais. Essa aplicação consiste de uma agenda com operações de busca, inserção e remoção de contatos.

A Tabela 5.8 apresenta as métricas físicas para uma execução de seis versões da aplicação *AddressBook*. Assim como nos experimentos com a aplicação mpegaudio, as diferentes versões desta aplicação foram obtidas pela refatoração da versão original (*AddressBook*), através da aplicação de sucessivas aplicações de *method inline*, para os métodos de maior consumo em ciclos.

Tabela 5.8: Métricas Físicas *AddressBook*.

	Ciclos	*Potência* (*mW*)	*Energia (mJ)*
AddressBook	10947	14,420720	0,031572724
AddressBookRef1	10849	14,294199	0,031015552

AddressBookRef2	9712	13,329553	0,025891324
AddressBookRe3	9388	12,904233	0,024228988
AddressBookRef4	9349	12,885126	0,024092608
AddressBookRef5	9316	12,853278	0,023948228

A Tabela 5.9 apresenta as métricas de qualidade de software para as diferentes versões da aplicação *AddressBook*. Assim como na Tabela 5.6, as linhas representam as métricas apresentadas no Capítulo 3 e as colunas representam as diferentes versões da mesma aplicação.

Tabela 5.9: Métricas de Qualidade de Software *AddressBook*.

	Address Book	*Address Book Ref1*	*Address Book Ref2*	*Address Book Ref3*	*Address Book Ref4*	*Address Book Ref5*
Abstração	0	0	0	0	0	0
Acoplamento Aferente	0	0	0	0	0	0
Acoplamento Eferente	0	0	0	0	0	0
Complexidade Ciclomática	37	13	13	13	11	10
Distância Normalizada da Sequência Principal	0	0	0	0	0	0
Falta de Coesão dos Métodos	0,393	0,1	0	0	0	0
Instabilidade	1	1	1	1	1	1
Métodos Ponderados por Classe	37	13	13	13	11	10
Número de Atributos	11	1	1	1	11	11
Número de Atributos Estáticos	0	0	0	0	0	0
Número de Classes	2	2	2	2	2	2
Número de Filhos	0	0	0	0	0	0

Número de Interfaces	0	0	0	0	0	0
Número de Linhas de Código dos Métodos	59	58	49	49	41	40
Número de Métodos	31	8	8	8	7	6
Número de Métodos Estáticos	2	1	1	1	1	1
Número de Métodos Sobrescritos	1	0	0	0	0	0
Número de Pacotes	1	1	1	1	1	1
Número de Parâmetros	25	6	6	6	5	4
Número total de Linhas de Código	108	95	83	83	68	65
Profundidade da Árvore de Herança	3	2	2	2	2	2
Profundidade de Blocos Aninhados	37	13	13	13	11	10

Finalmente, a Tabela 5.10 apresenta os erros de predição de ciclos e potência para as diferentes versões da aplicação *AddressBook*. Esses resultados são de interesse especial, pois acompanhando a coluna referente ao treinamento, cujo exemplo de teste é a versão *AddressBook*, pode-se notar que a rede neural e o modelo regressivo tem erros de predição bastante altos, pois os dados de predição fugiram aos intervalos de confiança dos métodos. Já na coluna seguinte da mesma tabela podemos notar que a rede neural teve um resultado muito mais próximo do real que o modelo regressivo.

Tabela 5.10: Erro de Predição para as Diferentes Implementações do *AddressBook*.

	Address Book	*Address BookRef1*	*Address BookRef2*	*Address BookRef3*	*Address BookRef4*	*Address BookRef5*
RNA (Ciclos)	-83,7153	4,43244	-3,93186	3,55017	9,94299	-7,69313
Regressão (Ciclos)	-150,2268	46,6129	-4,46332	5,192239	8,319161	-8,593404
RNA (Potência)	201,7673	-19,20132	6,68543	17,43552	-0,59927	0,60288
Regressão (Potência)	71,32059	-39,13496	-23,7904	-17,4182	-100	-100

Durante a realização dos experimentos apresentados nesta seção foram encontrados alguns resultados inesperados nos consumos de recursos físicos após uma série de aplicações da otimização *method inline*. Na seção que segue explicaremos o porquê desses resultados inesperados.

5.1.2 Sobrecarga das refatorações sobre a JVM

Como pode ser visto na Tabela 5.5, embora na maior parte das iterações de refatoração o desempenho da aplicação tenha melhorado, quando se observa o consumo de ciclos das versões mpegaudioRef3 e mpegaudioRef4 é possível notar que ao invés de um decréscimo nos ciclos consumidos pela aplicação houve um aumento dos mesmos.

Para entender o porquê desse aumento inesperado dos ciclos foi realizada uma análise dos histogramas das instruções executadas por estas duas versões da aplicação mpegaudio. A Tabela 5.11 apresenta um segmento do histograma de instruções dessas duas implementações onde destacamos instruções de interesse que variaram de uma execução para a outra. Na mesma tabela também é apresentada a diferença entre o número de ocorrências das instruções nas diferentes versões analisadas. No Apêndice C são apresentados os histogramas das duas versões, removendo as instruções com zero ocorrências no histograma, por motivos de espaço.

Tabela 5.11: Segmento do Histograma de Instruções das Versões mpegaudioRef3 e mpegaudioRef4.

	mpegaudio Ref3	mpegaudio Ref4	Diferença	Custo Unitário Ciclos	Custo Unitário Energia
iload	2,3E+09	2,4E+09	9,9E+07	4	45,2
iload_2	2,9E+08	1,9E+08	-1E+08	4	24,7
aload_0	9,2E+08	9,4E+08	2,3E+07	4	24,7
aload_1	2E+08	1,8E+08	-2E+07	4	24,7
istore	4E+08	4E+08	764100	6	67,42
istore_2	1,7E+07	1,6E+07	-764100	5	25,82
return	6336716	5572616	-764100	14	333,6
getfield	1,2E+09	1,2E+09	7,2E+07	1	151,4
invokevirtual	9589574	8825474	-764100	11	338,9

Uma breve análise da Tabela 5.11 nos revela dados interessantes. Entre as duas versões da aplicação podemos notar que o número de instruções de retorno (*return*) e de chamada (*invokevirtual*) de métodos diminuiu exatamente no mesmo número, o que indica o número de vezes que o método que sofreu *inline* fora chamado na versão mpegaudioRef3. No entanto, embora a redução dessas duas custosas instruções, o acréscimo no número de ocorrências da instrução *getfield*, responsável pelo acesso a

atributos de outros objetos, que também é uma instrução bastante custosa, causou um aumento nos ciclos da aplicação.

Além disso, ainda na Tabela 5.11, podemos notar que o número de instruções *iload* aumentou, enquanto as instruções *iload_2* diminuíram. Essas instruções são responsáveis pela carga de dados do *pool* de variáveis para o topo da pilha de operandos. A diferença entre essas duas instruções é que a instrução *iload_2* usa um índice implícito para acessar o *pool* de variáveis, carregando na pilha de operandos sempre a terceira posição do pool de variáveis, que deve ser do tipo inteiro. Já a instrução *iload* necessita de um operando explícito armazenado no topo da pilha de operandos, indicando qual posição do *pool* de variáveis ele deve colocar no topo da pilha de operandos. Esta última instrução, por ser um pouco mais complexa, necessita de mais energia para ser executada. De fato o aumento do consumo energético entre as implementações mpegaudioRef3 e mpegaudioRef4 se deve a esse tipo de trocas de instruções que o compilador teve de fazer para poder acoplar o escopo do método que sofreu *inline* no corpo, e o escopo do método ou métodos que o chamava antes do *inline*.

Já no caso das outras aplicações apresentadas na seção anterior, visto que elas são bem mais simples, há um balanço positivo entre a troca de instruções necessárias as chamadas de métodos, como *invokes* e *returns*, a sobrecarga por acesso a atributos de outros objetos, *getfield*, e as trocas de instruções de carga, com índice implícito (*iload_<n>*) por instruções com índice explícito (*iload*).

Também faz-se importante ressaltar que para outras linguagens e arquiteturas essas restrições não seriam encontradas, embora outras adversidades pudessem surgir.

6 ANÁLISE DOS RESULTADOS

Os resultados experimentais confirmam que decisões feitas na fase de projeto do software podem impactar as propriedades físicas do sistema final. Baseado nisso, foi proposto o uso de redes neurais artificiais para predizer as métricas físicas de sistemas embarcados a partir das métricas de qualidade do código fonte da aplicação.

Embora, como esperado, os resultados de predição das redes neurais tenham sido satisfatórios, para possibilitar uma melhor avaliação da qualidade das predições realizadas pela RNA também foram realizadas predições utilizando um modelo regressivo multivariado. Esses experimentos mostraram que mesmo em situações onde os dados variam a ponto de fugir ao intervalo de confiança do método de regressão multivariada, a RNA ainda foi capaz de predizer adequadamente as características físicas, embora a RNA em si também tenha o seu próprio intervalo de confiança para extrapolação de exemplos.

Além disso, é importante ressaltar que um extenso esforço foi realizado até que se alcançassem os melhores resultados apresentados. Durante esse processo de aprimoramento da abordagem fizeram-se necessários ajustes sobre a metodologia, de forma que os dados de entrada das redes neurais fossem simplificados, facilitando, assim, a tarefa de aprendizado das redes neurais. Dentre esses ajustes podemos citar: a aplicação da metodologia para várias versões de uma mesma aplicação, a normalização de dados de grande magnitude, o uso de refatoração para gerar um conjunto de treinamento a partir de uma versão funcional da aplicação.

Os resultados dos experimentos levaram, também, a importantes constatações sobre as características da máquina virtual Java, cuja arquitetura foi pensada em um contexto de linguagens orientadas a objetos com forte apego a boas práticas de programação, onde os métodos são curtos e com responsabilidades bem definidas.

Dessa forma, durante o processo incremental de refatorações da Seção 5.1.1, notou-se que, após algumas aplicações do *method inline*, os consumos de energia e ciclos aumentaram, ao invés de diminuir como era esperado. Isto ocorreu porque a JVM tem instruções que privilegiam escopos pequenos, realizando cargas (por exemplo, instruções de *iload_<n>*) das primeiras quatro do *pool* de variáveis, dentro de um mesmo escopo, utilizando um índice implícito, assim necessitando de menos acessos a memória. Então, quando métodos com muitas variáveis são implementados, essa sobrecarga pode ser pior que a utilização de chamadas a métodos (instruções de *invoke*).

No entanto, faz-se necessário ressaltar que para outras arquiteturas de processadores os limites encontrados para as refatorações seriam diferentes, e provavelmente mais

benéficos a aplicação das otimizações de código. Isto justifica o fato de o compilador Java não realizar otimizações durante o processo de geração dos binários (arquivos *class*) deixando a tarefa de otimização para a JVM.

7 CONCLUSÕES E TRABALHOS FUTUROS

7.1 Conclusões

Este trabalho demonstrou o potencial da abordagem de predição de consumo de uma aplicação baseado nas suas métricas de qualidade de software, obtidas a partir do código fonte da aplicação. A metodologia proposta demonstrou-se capaz de predizer de forma acurada aplicações de complexidade considerável, como o decodificador mpegaudio do conjunto de *benchmarks* Spec JVM 2008, mesmo quando a otimização aplicada durante os experimentos alterou os consumos da aplicação de forma inesperada devido a características específicas da arquitetura alvo.

Além disso, os pré-processamentos das métricas de qualidade de software, propostos no Capítulo 5, mostraram-se bastante eficientes na tarefa de aperfeiçoar a o treinamento da rede neural artificial, permitindo uma redução considerável dos erros de predição para exemplos não apresentados.

Com os resultados de predição alcançados os projetistas podem obter um retorno sobre alterações no código de forma rápida, sem precisar lançar mão de demorados processos de simulação. Portanto, esse trabalho contribui para a agilidade do processo de desenvolvimento de software para sistemas embarcados, pois modificações no código, sejam elas refatorações, otimizações ou mesmo pequenas extensões de funcionalidades, podem ter seu impacto avaliado de forma mais rápida que nas abordagens convencionais, baseadas em simulação, embora com menor precisão.

Além disso, os resultados alcançados levam a uma diversidade de possíveis trabalhos futuros, que são apresentados na seção que segue.

7.2 Trabalhos Futuros

Embora os erros de predição da RNA foram considerados satisfatórios o erro da predição para com o sistema real pode ser maior, visto que a ferramenta DESEJOS também tem um erro de predição associado a sua modelagem simplificada da sua arquitetura. Um possível trabalho futuro seria o uso de um simulador com uma modelagem mais fiel a arquitetura real, como o CACO-PS (BECK e CARRO, 2003), possibilitando alcançar predições ainda mais próximas dos consumos reais.

Uma atividade que poderia gerar ganhos substanciais de produtividade em projetos de desenvolvimento de software, utilizando a abordagem proposta, seria a substituição do uso de um perfil de execução, na detecção das classes efetivamente utilizadas, por uma ferramenta capaz de produzir essa informação sem a necessidade de uma versão

funcional do código. Isto, associado a um desenvolvimento baseado em componentes previamente caracterizados quanto a seu consumo dos recursos físicos, possibilitaria ao projetista acessar uma estimativa de consumo em etapas iniciais do projeto, talvez ainda na etapa de modelagem da aplicação. Nesse contexto, a abordagem de predição proposta nesse trabalho associada a uma biblioteca de componentes caracterizada, poderia ser aplicada em níveis mais altos de abstração, visto que algumas das métricas aqui discutidas são diretamente obtidas a partir do nível de modelos.

Para resolver o problema do uso de perfis de execução, uma alternativa, que ainda não foi avaliada, seria o emprego de *call graphs* para a predição de quais classes estão realmente sendo utilizadas no projeto, dispensando assim custosos processos de simulação da execução do código na plataforma alvo.

Ainda no contexto dos objetivos gerais deste trabalho, trabalhos futuros incluem a análise do emprego de outros tipos de refatorações, ou mesmo otimizações, em aplicações orientadas a objeto, bem como o impacto nas métricas físicas causado por essas alterações nas aplicações.

Se outros tipos de modificações nos códigos de aplicações também puderem ter seu comportamento predito com baixos erros, então, tornar-se-á possível avançar na pesquisa propondo a construção de uma ferramenta capaz de, baseado nas predições da RNA, escolher dentre um conjunto de otimizações qual seria a melhor modificação a ser realizada no código de forma a gerar uma aplicação para melhor uso de um determinado recurso físico. Em outras palavras, uma ferramenta automática de exploração de espaço de projeto que seja capaz de realizar automaticamente diferentes tipos de modificações no código fonte da aplicação, de forma a buscar por diferentes versões da mesma, mas que possam apresentar consumos de potência, memória ou ciclos mais interessantes para o projeto. Inclusive, a partir do uso de várias RNAs a ferramenta poderia realizar a exploração do espaço de projeto em busca de uma otimização multivariada dos recursos, por exemplo, minimizar o consumo de ciclos e memória.

Outra atividade que também poderia complementar este trabalho consiste na análise das relações entre métricas de baixo nível e métricas de alto nível em outras arquiteturas de processadores.

Além disso, a análise do uso de outras linguagens orientadas a objetos, para facilitar a obtenção dessas relações, visto que a linguagem Java necessitaria de um sistema operacional e uma JVM para outros tipos de arquiteturas.

Ainda, uma alternativa a exploração do impacto de todas as métricas de software para com todas as métricas físicas seria o uso de um conjunto reduzido de métricas para cada característica física. Assim, um conjunto de métricas de alto nível utilizado para predizer os custos de energia poderia ser diferente do conjunto utilizado para predizer consumo de memória. Dessa forma, poderíamos facilitar o aprendizado de métodos de aprendizado de máquina, retirando da entrada características que causam pouco, ou nenhum, impacto direto no consumo de uma determinada métrica física.

REFERÊNCIAS

ABNT. **NBR ISO 8402. Gestão da qualidade e garantia da qualidade - Terminologia.** Associação Brasileira de Normas Técnicas. Rio de Janeiro. 1994.

AGGARWAL, K. K. et al. Empirical Study of Object-Oriented Metrics. **Journal of Object Technology**, v. 5, 2006.

AMBLER, S. **Análise e projeto orientado a objeto:** seu guia para desenvolver sistemas robustos com tecnologia de objetos. [S.l.]: Infobook, 1998.

BALASUBRAMANIAN, N. V. **Object-Oriented Metrics**. [S.l.]: IEEE Computer Society. 1996. p. 30--.

BECK, A. C. S.; CARRO, L. **CACO-PS:** A General Purpose Cycle-Accurate Configurable Power Simulator. Proceedings of the 16th Symposium on Integrated Circuits and System Design. São Paulo: ACM Press. 2003. p. 349 - 354.

BOEHM, B. W. **Software Engineering Economics**. [S.l.]: Prentice Hall, 1981.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **IEEE Trans. Softw. Eng.**, v. 20, 1994.

CONTE, S. D.; DUNSMORE, H. E.; SHEN, V. Y. **Software engineering metrics and models**. [S.l.]: Benjamin-Cummings Publishing Co., Inc., 1986.

CORMEN, T. H. et al. **Algoritmos:** Teoria e Prática. [S.l.]: Editora Campus, 2001.

CYBENKO, G. Approximation by superpositions of a sigmoidal function. **Mathematics of Control, Signals, and Systems (MCSS)**, v. 2, 1989.

FENTON, S. L.; PFLEEGER, N. E. **Software Metrics:** A Rigorous and Practical Approach. 1. ed. [S.l.]: Boston : PWS : London : International Thomson Computer Press, 1996. ISBN 1850322759.

FERREIRA, R. R. **Automatic code generation and solution estimate for object-oriented embedded software**. [S.l.]: ACM. 2008. p. 909-910.

FURSIN, G. et al. **MiDataSets:** creating the conditions for a more realistic evaluation of Iterative optimization. [S.l.]: [s.n.]. 2007.

FURSIN, G. et al. **MILEPOST GCC:** machine learning based research compiler. [S.l.]: [s.n.]. 2008.

FURSIN, G.; COHEN, A. **Building a practical iterative interactive compiler**. [S.l.]: [s.n.]. 2007.

GAMMA, E. et al. **Design patterns:** elements of reusable object-oriented software. [S.l.]: Addison-Wesley Professional, 1995.

GANSSLE, J. G. **The Art of Designing Embedded Systems**. [S.l.]: Newnes, 2008.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded Software Engineering: The State of the Practice. **IEEE Software**, v. 20, 2003.

GUTHAUS, M. R. et al. **MiBench:** A free, commercially representative embedded benchmark suite. [S.l.]: [s.n.]. 2001.

HALSTEAD, M. H. **Elements of Software Science, Operating, and Programming**. New York, NY: Elsevier, 1977.

HAYKIN, S. **Neural Networks:** A Comprehensive Foundation. [S.l.]: Macmillan, 1994.

HENDERSON-SELLERS, B. **Object-oriented metrics:** measures of complexity. [S.l.]: Prentice-Hall, Inc., 1996.

HITZ, M.; MONTAZERI, B. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. **IEEE Transactions on Software Engineering**, v. 22, 1996.

HOGAN, J. **An Analysis of OO Software Metrics**. [S.l.]. 1997.

IEEE. **IEEE standard for a software quality metrics methodology**. Institute of Electrical and Electronic Engineers. New York, p. 32. 1998.

INTEL. **Intel Atom Processor E6x5C Series-Based Plataform for Embedded Computing**, 2010. Disponivel em: <http://edc.intel.com/Link.aspx?id=3961>. Acesso em: dezembro 2010. http://edc.intel.com/Link.aspx?id=3961.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design & Test**, v. 5 , n. 18, Setembro 2001.

JERRAYA, A. A. et al. **Embedded Software for Soc**. [S.l.]: Kluwer Academic Publishers, 2003.

JONES, C. **Programming Productivity**. [S.l.]: McGraw-Hill, 1986.

KAFURA, D. **A survey of software metrics**. [S.l.]: ACM. 1985. p. 502-506.

KAN, S. H. **Metrics and Models in Software Quality Engineering**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

KANER, C.; BOND, W. P. **Software Engineering Metrics:** What Do They Measure and How Do We Know? [S.l.]: [s.n.]. 2004.

KOENIG, A. Patterns and Antipatterns. **Journal of Object Oriented Programming**, v. 8, 1995.

LEDGARD, H. F.; MARCOTTY, M. A genealogy of control structures. **Commun. ACM**, v. 18, 1975.

LIMA, E. L. O teorema de Euler sobre Poliedros. **Revista Matemática Universitária**, n. 2, p. 57-74, 1985.

LORENZ, M.; KIDD, J. **Object-oriented software metrics:** a practical guide. [S.l.]: Prentice-Hall, Inc., 1994.

LYNCH, W. C.; BROWNE, J. C. Performance Evaluation: A Software Metrics Success Story. In: PERLIS, A. J.; SAYWARD, F.; SHAW, M. **Software metrics:** an analysis and evaluation. [S.l.]: MIT Press, 1981. Cap. 10, p. 171-183.

MANGALAMPALLI, A.; JAIN, A. K. Correlation of Inheritance-Based Object-Oriented Metrics and Low-Level Metrics. **ICFAI Journal of Systems Management**, v. 5, p. 7-15, 2007.

MARTIN, R. C. **Designing object-oriented C++ applications:** using the Booch method. [S.l.]: Prentice-Hall, Inc., 1995.

MARTIN, R. C. **Agile Software Development, Principles, Patterns, and Practices**. [S.l.]: Prentice-Hall, Inc, 2002.

MARWEDEL, P. **Embedded System Design**. 1. ed. Dortmund: Kluwer Academic Publishers, 2003.

MATTOS, J. C. B.; CARRO, L. **Object and Method Exploration for Embedded Systems Applications**. Proc. of the Symposium on Integrated Circuits and Systems Design (SBCCI). Rio de Janeiro, Brazil: ACM Press. 2007.

MCCABE, J. A Complexity Measure. **IEEE Transactions on Software Engineering**, 4, Dezembro 1976. 308-320.

METRICS. **Eclipse Metrics Plug-in**, 2005. Disponivel em: <http://metrics.sourceforge.net/>. Acesso em: 1 dezembro 2010.

MITCHELL, T. M. **Machine Learning**. [S.l.]: McGraw-Hill, 1997.

MÖLLER, K.-H.; PAULISH, D. J. **Software Metrics:** A Practitioner's Guide to Improved Product Development. [S.l.]: IEEE Computer Society Press, 1992.

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, 8, Abril 1965. 114–117.

MUSA, J. D. A Theory of Software Reliability and Its Application. **IEEE Trans. Software Eng.**, v. 1, 1975.

PRESSMAN, R. S. **Engenharia de Software**. [S.l.]: McGraw-Hill, 2002.

PUTNAM, L. H. A General Empirical Solution to the Macro Software Sizing and Estimating Problem. **IEEE Trans. Softw. Eng.**, v. 4, 1978.

REDIN, R. et al. On the Use of Software Quality Metrics to Improve Physical Properties of Embedded Systems. In: KLEINJOHANN, B.; WOLF, W.; KLEINJOHANN, L. **Distributed Embedded Systems:** Design, Middleware and Resources. [S.l.]: Springer Boston, v. 271, 2008. p. 101-110.

RIEGER, M. et al. **Refactoring for Performance:** An Experience Report. [S.l.]: [s.n.]. 2007. p. 206-214.

ROSENBERG, L. H. **Applying and Interpreting Object Oriented Metrics**. [S.l.]: NASA Software Assurance Technology Center (SACT). 1998.

RYAN, C.; ROSSI, P. Software, Performance and Resource Utilisation Metrics for Context-Aware Mobile Applications. **Software Metrics, IEEE International Symposium on**, v. 0, 2005.

SELIC, B. UML 2: a model-driven development tool. **IBM Systems Journal**, v. 45, 2006.

SMAALDERS, B. Performance Anti-Patterns. **Queue**, v. 4, 2006.

SOMMERVILLE, I. **Software engineering**. 8. ed. [S.l.]: Addison Wesley, 2006. 864 p. ISBN 0321313798.

SPEC, Spec JVM 2008. **Spec JVM 2008 Benchmarkset**, 2010. Disponivel em: <www.spec.org>. Acesso em: dezembro 2010.

SRINIVASAN, J.; DOBRIN, R.; LUNDQVIST, K. **'State of the Art' in Using Agile Methods for Embedded Systems Development**. 33rd Annual IEEE International Computer Software and Applications Conference, 2009. COMPSAC '09. Seattle, WA: [s.n.]. 2009. p. 522-527.

SYNOPSYS. **Synopsys Power Compiler**, 2010. Disponivel em: <http://www.synopsys.com/>. Acesso em: dezembro 2010.

VERKAMO, A. I. et al. **Design patterns in performance prediction**. [S.l.]: ACM. 2000. p. 143-144.

WELCH, T. A. A Technique for High-Performance Data Compression. **Computer**, v. 17, 1984.

XENOS, M. et al. **Object-oriented metrics - a survey**. [S.l.]: Federation of European Software Measurement Associations. 2000.

YANG, L. et al. **An Evaluation of Confidence Bound Estimation Methods for Neural Networks**. [S.l.]: Kluwer, B.V. 2002. p. 71-84.

ZHANG, W.; JARZABEK, S. Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices. In: OBBINK, H.; POHL, K. **Software Product Lines**. [S.l.]: Springer Berlin / Heidelberg, v. 3714, 2005. p. 57-69.

ZUSE, H. **Software complexity:** measures and methods. [S.l.]: Walter de Gruyter \& Co., 1991.

APÊNDICE A GLOSSÁRIO DE TERMOS RELATIVOS A MÉTRICAS DE SOFTWARE (IEEE, 1998)

Atributo: uma propriedade física ou abstrata que possa ser mensurada.

Intervalo crítico: valores de métricas usados para classificar o software nas categorias: aceitável, marginal (próximo ou no limite aceitável) ou inaceitável.

Valor crítico: valor de métrica de uma métrica validada que é usado para identificar softwares que tem qualidade inaceitável.

Métrica direta: uma métrica é direta se não depende de uma medida de qualquer outro atributo, ou seja, se ela é independente de outros atributos.

Valor de métrica direta: um objetivo numérico para um fator de qualidade a ser alcançado no produto final. Por exemplo, tempo médio entre falhas é uma métrica direta da confiabilidade do produto final.

Medida: (a) é o número ou símbolo atribuído a uma entidade por um mapeamento de forma a caracterizar um atributo. (b) Aplicar uma métrica.

Medição: o ato ou processo de atribuir um número ou categoria a uma entidade para descrever um atributo desta entidade.

Amostra de métrica: um conjunto de valores de métrica que é extraído do banco de dados de métricas e usado na validação de métricas.

Validação de métricas: o ato ou processo de garantir que uma métrica prediz ou avalia de forma confiável um fator de qualidade.

Valor de métrica: uma saída de métrica ou um elemento que é do intervalo da métrica.

Métrica preditiva: uma métrica aplicada durante o desenvolvimento e usada para predizer valores de um fator de qualidade de software.

Valor de métrica preditiva: um objetivo numérico relacionado a um fator de qualidade a ser alcançado durante o desenvolvimento do sistema. Este é um requisito intermediário anterior ao indicador do desempenho do sistema final. Por exemplo, erros de projeto ou código podem ser predições da confiabilidade do sistema final.

Métrica de processo: uma métrica usada para medir características dos métodos, técnicas e ferramentas empregadas no desenvolvimento, implementação e manutenção do sistema de software.

Métrica de produto: Uma métrica usada para medir as características de um produto de software intermediário ou final durante o processo de desenvolvimento de software.

Atributo de qualidade: Uma característica do software ou um termo genérico aplicado a fatores de qualidade, subfatores de qualidade ou valores de métricas.

Fator de qualidade: um atributo gerenciável de software que contribui para sua qualidade.

Amostra de fator de qualidade: um conjunto de valores de fatores de qualidade que é retirado do banco de dados de métricas de qualidade e usado na validação de métricas.

Valor de fator de qualidade: um valor de uma métrica direta que representa um fator de qualidade.

Subfator de qualidade: a decomposição de um fator ou subfator de qualidade para seu componente técnico.

Requisito de qualidade: um requisito que um atributo de software precisa apresentar para satisfazer um contrato, padrão, especificação, ou outra imposição formalmente documentada.

Componente de software: um termo geral usado para referenciar um sistema ou elemento de software, como um módulo, uma unidade, dados ou documentos.

Métrica de qualidade de software: uma função cujas entradas são dados sobre um software e cuja saída é um único valor numérico que pode ser interpretado como o grau em que o software possui um dado atributo que afeta sua qualidade.

Métrica validada: uma métrica cujos valores foram estaticamente associados com correspondentes valores de fatores de qualidade.

APÊNDICE B PARÂMETROS DE TREINAMENTO DAS REDES NEURAIS UTILIZADAS NOS EXPERIMENTOS

Neste apêndice serão apresentados os parâmetros de treinamento das redes neurais artificiais utilizadas nos experimentos apresentados nos Capítulos 4 e 5. Os experimentos foram realizados utilizando o *toolbox* de redes neurais do Matlab®. Para possibilitar a reprodução dos experimentos serão apresentados os parâmetros que precisaram ser ajustados, diferenciando-se dos valores padrão do *toolbox*, para que as redes neurais fossem capazes de aprender os dados apresentados.

B.1 Experimentos do Capítulo 4

A rede neural utilizada para os experimentos da Seção 4.1.1 utilizou os parâmetros pré-definidos no *toolbox* de redes neurais do Matlab®, com exceção dos apresentados na tabela a seguir.

Tabela B.1: Parâmetros de Treinamento da RNA utilizada na Seção 4.1.1.

Parâmetros de Treinamento	*Valor Aplicado*
Número de Épocas	10000000
Erro Médio Quadrático Máximo	5,00E-03
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	5
Num. Neurônios Camada Saída	4
Algoritmo de Aprendizado	Levenberg-Marquardt

Posteriormente, ainda no Capítulo 4, os dados das duas aplicações (MP3 e Controle da Cadeira de Rodas) foram separados para serem apresentados a duas redes neurais diferentes. A RNA responsável pelo aprendizado dos consumos da aplicação MP3 na Seção 4.1.2 são apresentados na

Tabela B.2.

Tabela B.2: Parâmetros de Treinamento da RNA utilizada, na Seção 4.1.2, para a aplicação MP3.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	10000
Erro Médio Quadrático Máximo	1,00E-05
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	3
Num. Neurônios Camada Saída	4
Algoritmo de Aprendizado	Levenberg-Marquardt

Ainda na Seção 4.1.2, outra rede neural foi utilizada, para o aprendizado das características físicas do algoritmo de controle de uma cadeira de rodas, e seus parâmetros são apresentados na Tabela B.3.

Tabela B.3: Parâmetros de Treinamento da RNA utilizada, na Seção 4.1.2, para a aplicação Controle de uma Cadeira de Rodas.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	10000
Erro Médio Quadrático Máximo	1,00E-05
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	5
Num. Neurônios Camada Saída	4
Algoritmo de Aprendizado	Levenberg-Marquardt

B.2 Experimentos do Capítulo 5

No Capítulo 5 as RNAs foram utilizadas para aprender uma única característica física de cada aplicação. A seguir são apresentados os parâmetros de treinamento das redes neurais cujos erros percentuais, para padrões não apresentados no treinamento,

são exibidos na Tabela 5.4. Na Tabela B.4 são apresentados os parâmetros para a RNA utilizada para aprender o consumo energético da aplicação DVM.

Tabela B.4: Parâmetros de treinamento para as RNAs utilizadas para predizer a energia consumida pela DVM.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	1000
Erro Médio Quadrático Máximo	5,00E-05
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	20
Num. Neurônios Camada Saída	1
Gradiente Mínimo	1E-3000
Decremento no “mu”	1E-20
“mu” inicial	1E-10
“mu" máximo	1E+300
Algoritmo de Aprendizado	Levenberg-Marquardt

Já para as RNAs responsáveis pelo aprendizado dos ciclos consumidos pelas versões da DVM os parâmetros de treinamento são apresentados na Tabela B.5.

Tabela B.5: Parâmetros de treinamento para as RNAs utilizadas para predizer os ciclos para as diferentes versões da DVM.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	1000
Erro Médio Quadrático Máximo	1,00E-05
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	20
Num. Neurônios Camada Saída	1
Gradiente Mínimo	9E-299
Decremento no “mu”	1E-2
“mu” inicial	1E-60

“mu" máximo	9E+299
Algoritmo de Aprendizado	Levenberg-Marquardt

Para os experimentos com a aplicação mpegaudio os parâmetros de treinamento da rede neural responsável pela predição de ciclos são apresentados na Tabela B.6.

Tabela B.6: Parâmetros de Treinamento das Redes Neurais para Predição de Ciclos das Versões da Aplicação Spec mpegaudio.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	10000
Erro Médio Quadrático Máximo	0,00
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	25
Num. Neurônios Camada Saída	1
Gradiente Mínimo	1E-3000
Decremento no “mu”	1E-2
“mu” inicial	1E-60
“mu" máximo	1E+60
Algoritmo de Aprendizado	Levenberg-Marquardt

Ainda sobre a aplicação mpegaudio, a rede neural para a predição de energia tem seus parâmetros apresentados na Tabela B.7.

Tabela B.7: Parâmetros de Treinamento das Redes Neurais para Predição de Energia das Versões da Aplicação Spec mpegaudio.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	10000
Erro Médio Quadrático Máximo	1,00E-05
Num. Neurônios Camada Entrada	23

Num. Neurônios Camada Escondida	25
Num. Neurônios Camada Saída	1
Gradiente Mínimo	1e-3000
Decremento no “mu”	1e-2
“mu” inicial	1e-60
“mu" máximo	1e+60
Algoritmo de Aprendizado	Levenberg-Marquardt

Na Tabela B.8 são apresentados os parâmetros das redes neurais utilizadas para predizer os custos de energia e ciclos da aplicação AddressBook, embora essas características físicas tenham sido preditas em redes neurais diferentes, os parâmetros utilizados foram os mesmos.

Tabela B.8: Parâmetros de Treinamento das Redes Neurais para Predição de Energia e Ciclos das Versões da Aplicação AddressBook.

Parâmetros de Treinamento	*Valor*
Número Máximo de Épocas	10000
Erro Médio Quadrático Máximo	1,00E-05
Num. Neurônios Camada Entrada	23
Num. Neurônios Camada Escondida	15
Num. Neurônios Camada Saída	1
Gradiente Mínimo	1e-3000
Decremento no “mu”	1e-2
“mu” inicial	1e-6
Algoritmo de Aprendizado	Levenberg-Marquardt

APENDICE C HISTOGRAMAS DE INSTRUÇÕES DAS APLICAÇÕES ANALISADAS NA SEÇÃO 5.1.1

Neste apêndice os gráficos de histogramas de instruções representativos das execuções realizadas para os experimentos da Seção 5.1.1. Nesses histogramas são apresentadas as instruções que apresentaram desvio padrão diferente de zero entre todas as versões das aplicações. Além disso, nesses histogramas ordenamos as instruções por número de ocorrências.

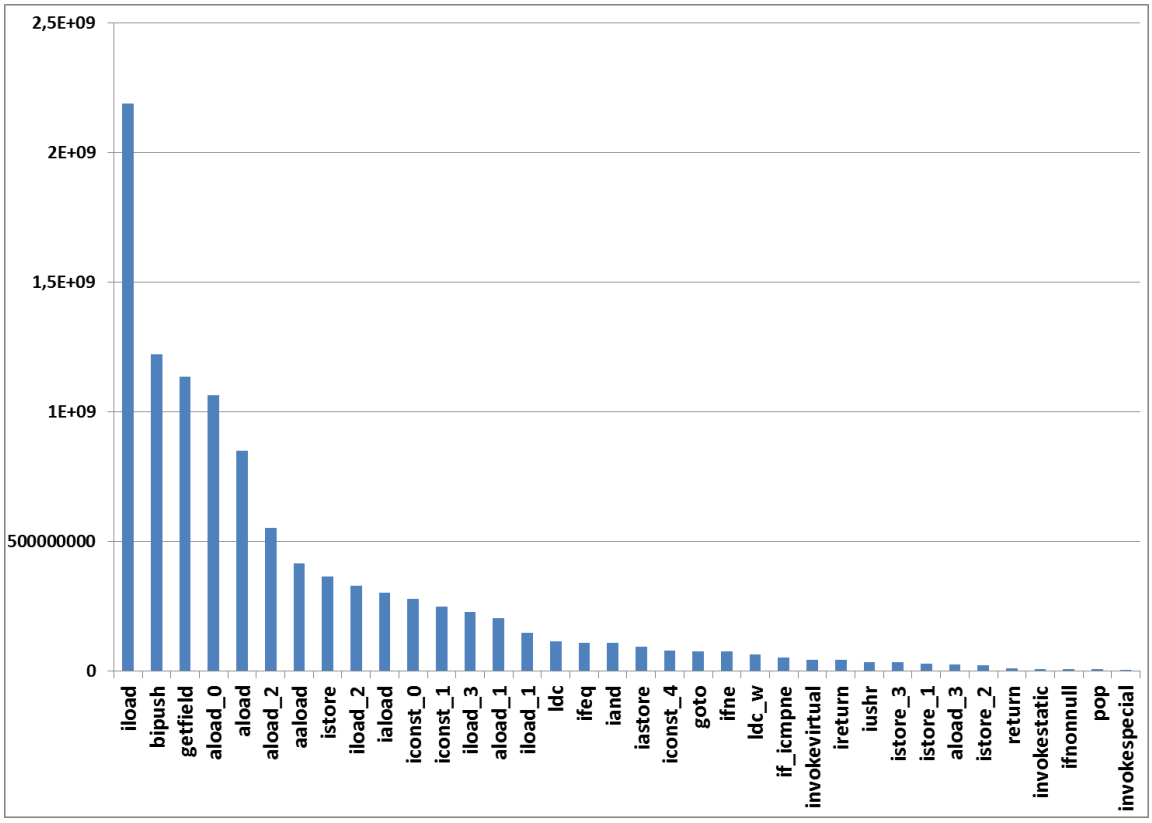

Figura C.1: Fragmento de histograma mpegaudio.

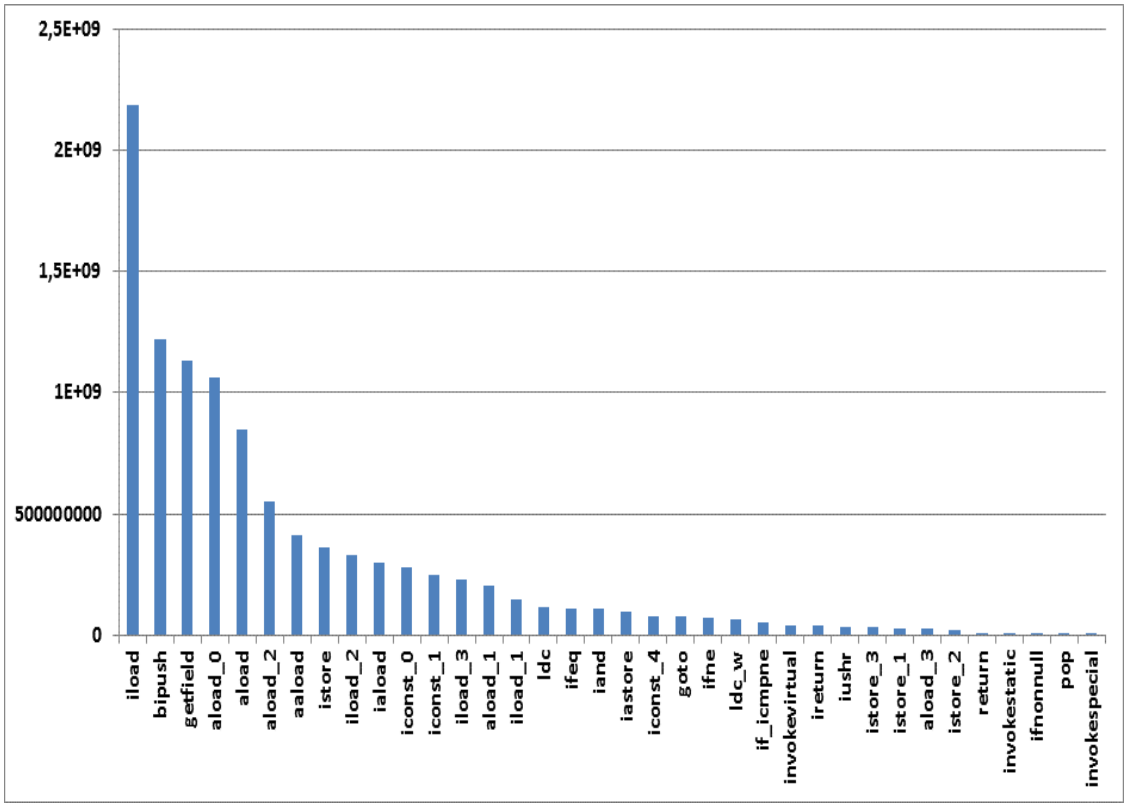

Figura C.2: Fragmento de histograma mpegaudioRef1.

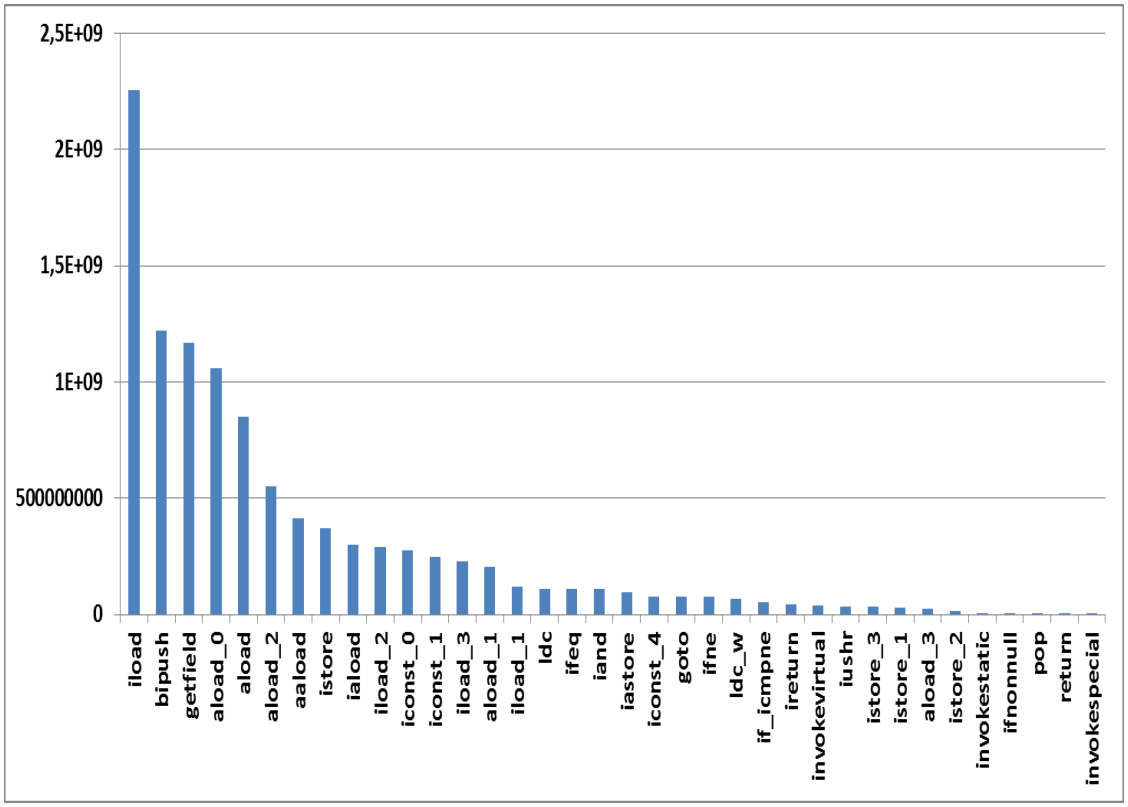

Figura C.3: Fragmento de histograma mpegaudioRef2.

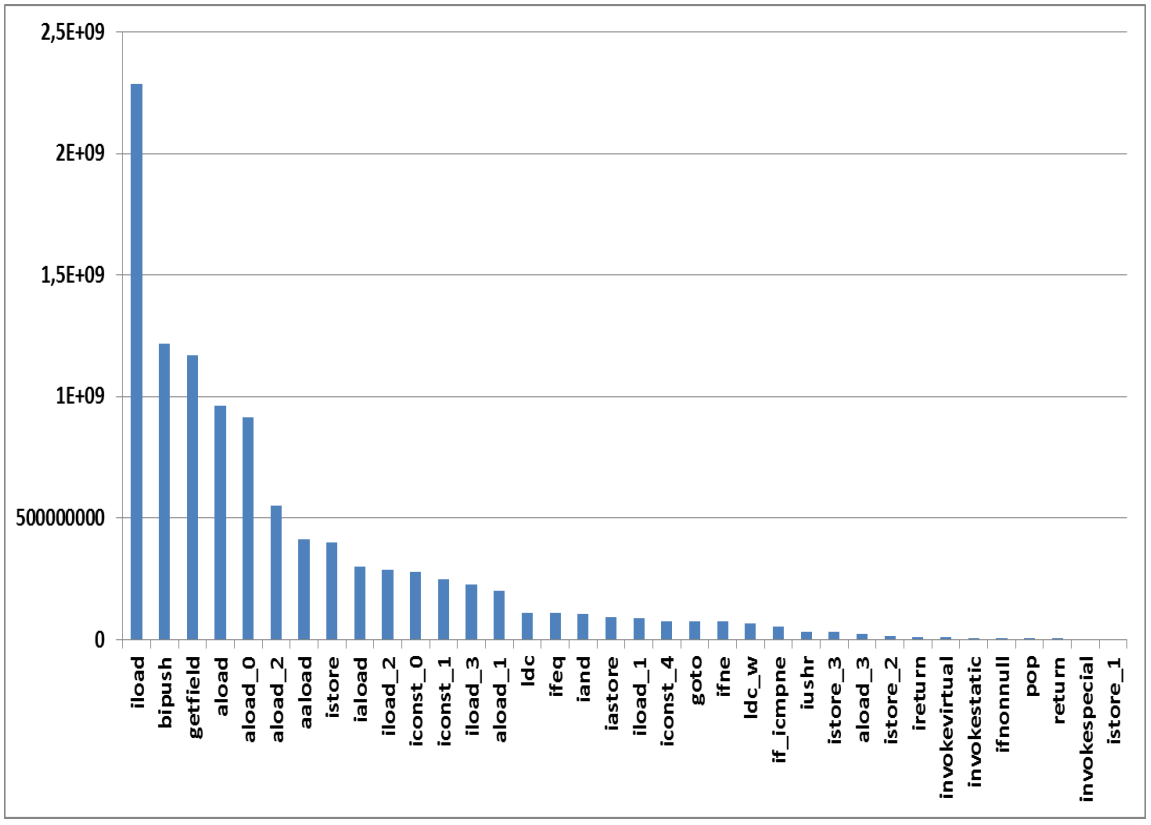

Figura C.4: Fragmento de histograma mpegaudioRef3.

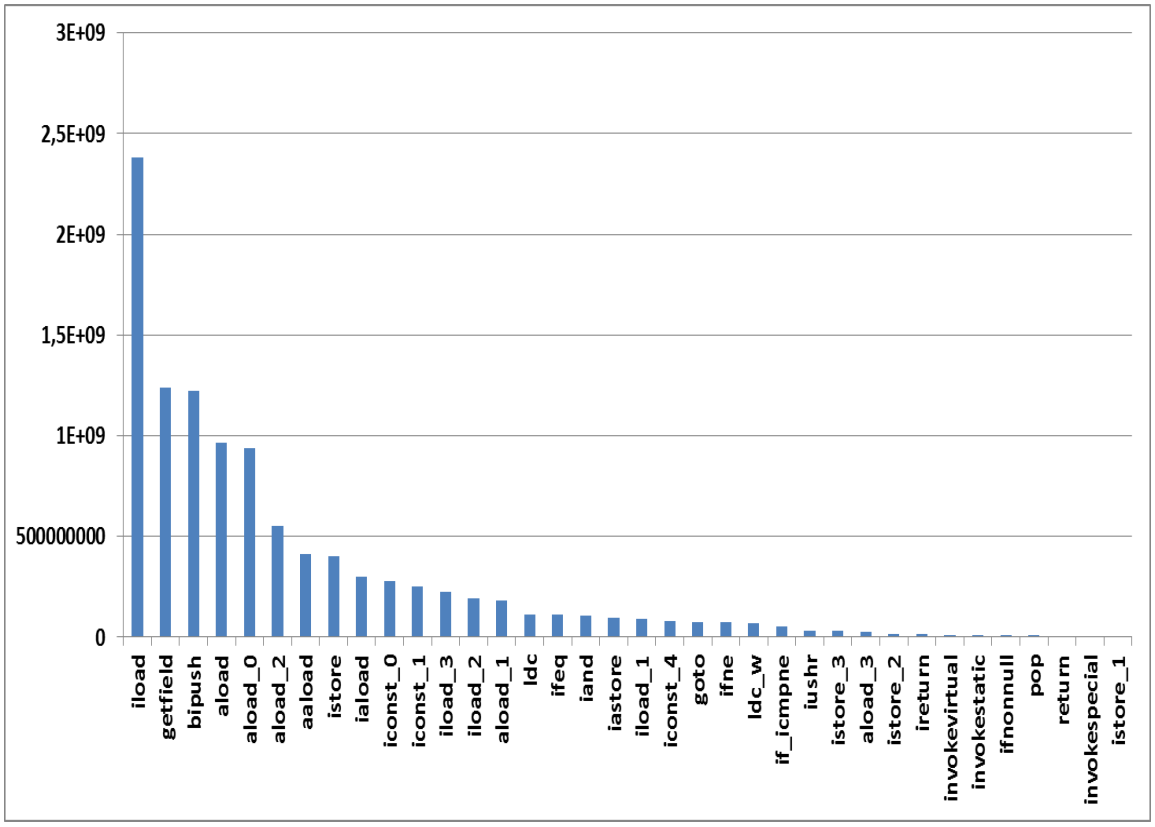

Figura C.5: Fragmento de histograma mpegaudioRef4.

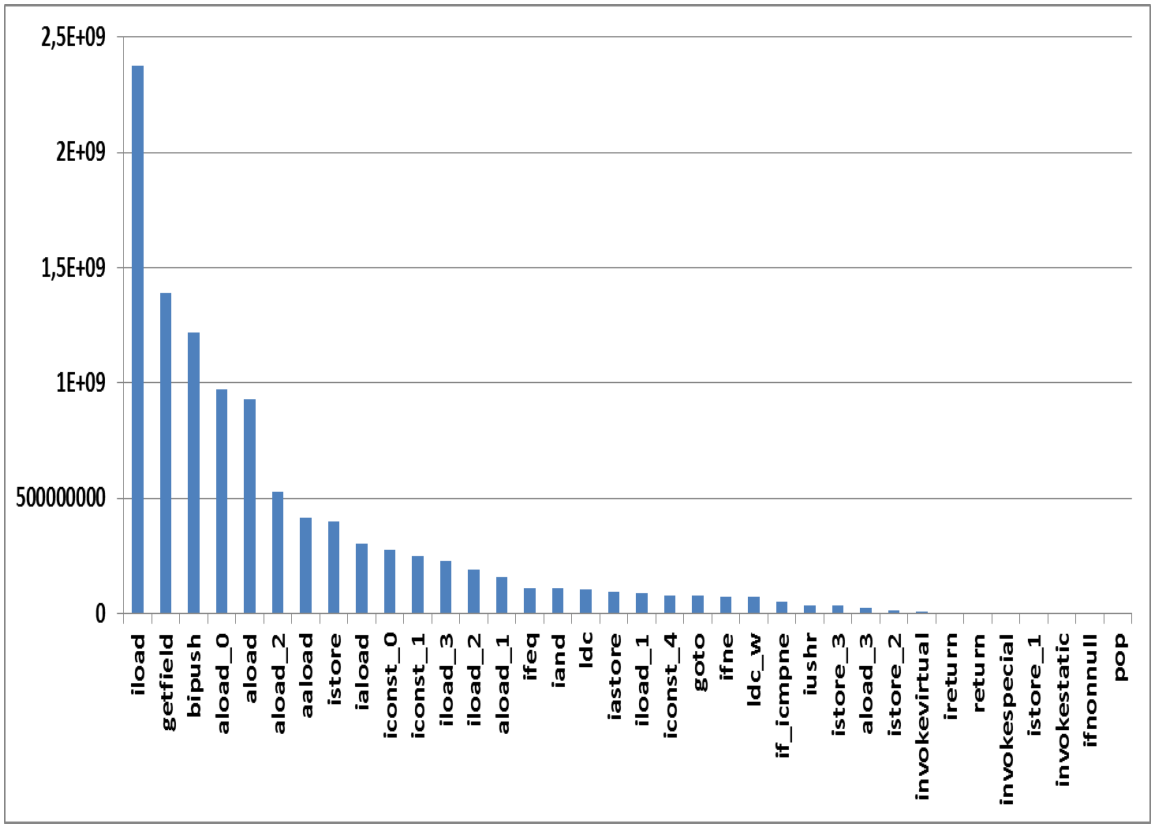

Figura C.6: Fragmento de histograma mpegaudioRef5.

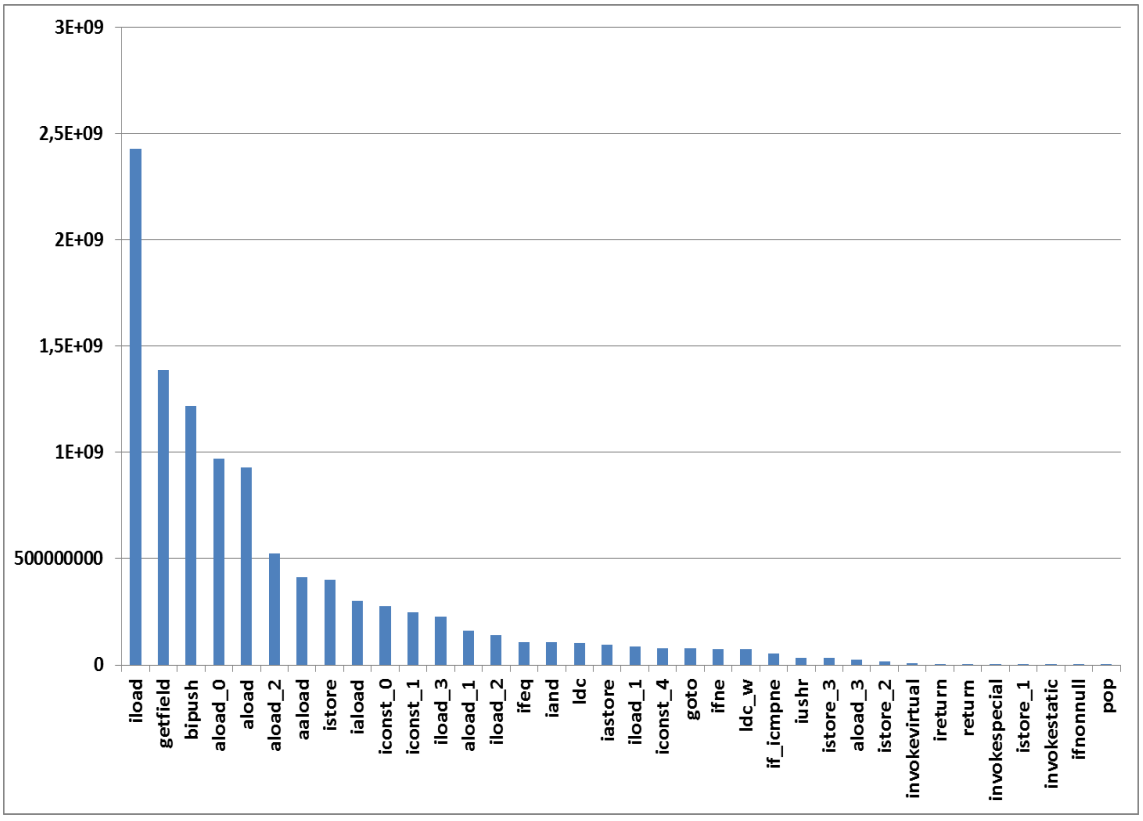

Figura C.7: Fragmento de histograma mpegaudioRef6.

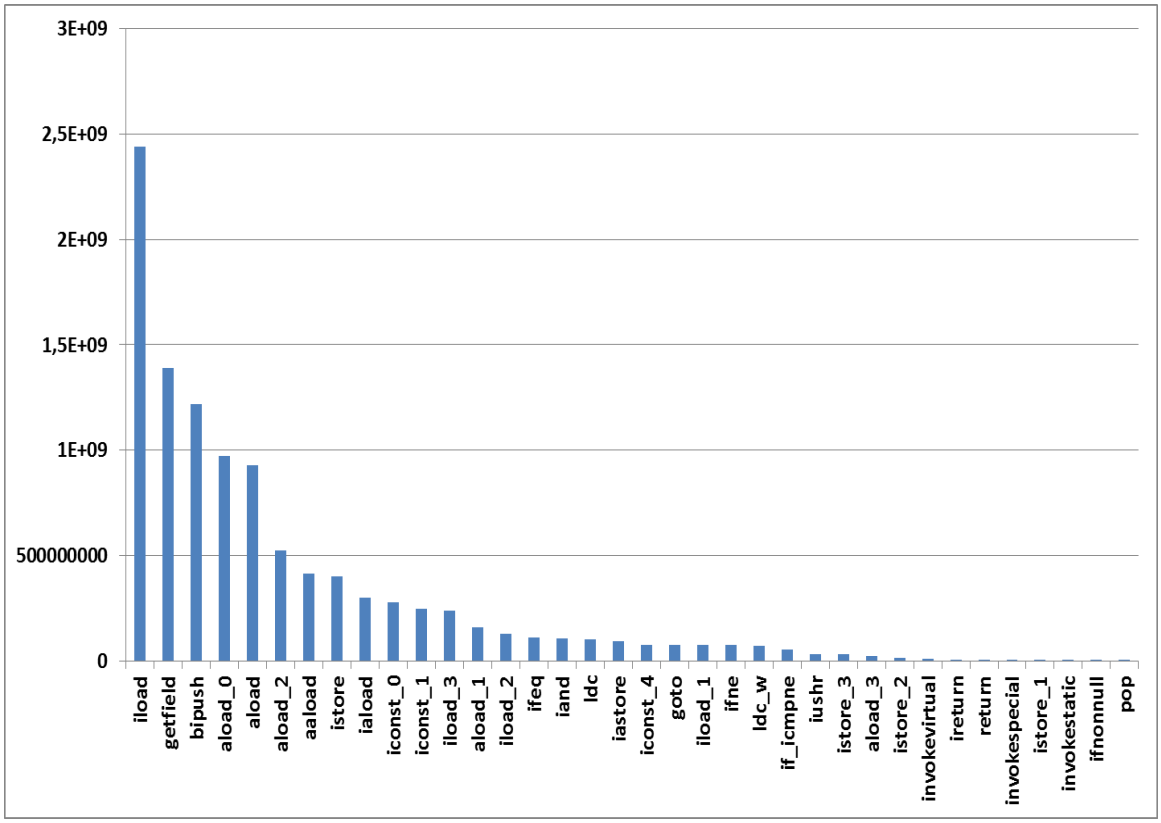

Figura C.8: Fragmento de histograma mpegaudioRef7.