Métricas OO Aplicadas a Código Objeto Java

Carla Tiaki Utsunomiya
Universidade Federal do Paraná (UFPR)
Curitiba - PR - Brasil
carlatiaki@bol.com.br

Márcio Delamaro
Edmundo Sérgio Spoto
Fundação Eurípides Soares da Rocha
Marília - SP - Brasil
{delamaro,dino}@fundanet.br

Resumo

Na busca de melhorias no processo de desenvolvimento de software para a obtenção de um produto de qualidade, várias métricas têm sido propostas, com as quais pode-se gerenciar este processo e detectar falhas de projeto. As métricas de software auxiliam na coleta de informação, fornecendo dados qualitativos e quantitativos sobre o processo e o produto de software. Elas identificam onde os recursos são necessários, constituindo assim importante fonte de informação para a tomada de decisão. Métricas podem ser aplicadas em diversas fases do desenvolvimento e em diversos produtos intermediários como especificação de requisitos, projeto ou código fonte. Este trabalho mostra a factibilidade de coletar-se algumas métricas de software a partir do código objeto (bytecode) Java. Tal abordagem pode ser útil em atividades como: teste de programas que utilizam componentes de terceiros, re-engenharia e outras nas quais não se tenha acesso ao código fonte. As métricas coletadas a partir do bytecode Java foram aplicadas em um estudo de caso com dois sistemas onde procurou-se relacionar as métricas com a propensão a falhas.

Palavras-chave: Métricas, Sistema Orientado a Objetos, Código Objeto, Propensão à Ocorrência de Falha.

Abstract

Searching for improvements in the software development process and aiming at a product with high quality, several metrics, that help to manage the software process and to detect project flaws, have been proposed. Software metrics provide qualitative and quantitative about the software process and the software product. The identify where resources should be allocated, being an important source for decision making. They can be applied in any phase of the development and on different intermediary products as requirement specification, design or source code. This work shows the feasibility of collecting some software metrics directly form Java object programs (Java bytecode). Such an approach can be useful in activities like: testing of third party components, re-engineering and others activities where the source code is not available. This approach has been applied in a case study to two Java systems, trying to relate the metrics to the existence of faults.

Keywords: *Metrics, Object-Oriented System, Source Object and Probability of Fault Detection.*

1 Introdução

O desenvolvimento de um sistema de software requer tempo e utilização de recursos. Desta forma, para a automação das atividades de desenvolvimento de um software de qualidade é fundamental a geração de informações precisas para se alocar recursos e tempos adequados no processo de desenvolvimento de software. [1].

As métricas de software auxiliam este processo de coleta de informações fornecendo dados qualitativos e quantitativos sobre o processo e o produto de software. Elas identificam onde os recursos são necessários e são fontes cruciais de informações para a tomada de decisão [12]. Teste de grandes sistemas pode ser citado como um exemplo de uma atividade que consome tempo e recursos. A capacidade de identificar antecipadamente módulos com propensão a falhas pode propiciar economia no desenvolvimento permitindo que os esforços de teste/verificação sejam concentrados nestes módulos [11].

Métricas de software existem desde a década de 60, como por exemplo as medidas do tamanho de um produto em linhas de código ou o índice Fog [10]. São aplicadas em todas as fases do desenvolvimento e aos mais variados produtos deste processo, como especificação, projeto ou código. Com o aparecimento e popularização do paradigma de desenvolvimento orientado a objetos (OO), surgiram também diversas métricas que visam avaliar características relacionadas a aspectos particulares da OO, como herança ou composição.

Este trabalho propõe a aplicação das principais métricas OO, propostas e validadas na literatura, em um contexto ligeiramente diferente daquele em que são geralmente aplicadas. Em geral, as métricas OO são coletadas a partir do código fonte ou do projeto de classes do software. Neste artigo, descreve-se como tais métricas podem ser obtidas a partir do código objeto (bytecode) Java e como elas foram implementadas numa ferramenta chamada JaBUTi. Como estudo de caso, foram utilizados dois sistemas produzidos em Java, nos quais as métricas foram coletadas a partir do bytecode. Procurou-se então relacionar as métricas com a propensão a falhas.

Na próxima seção são comentados os principais trabalhos relacionados, destacando-se aqueles que propõem ou avaliam as métricas OO. Na Seção 3 é descrita a ferramenta JaBUTi e a maneira como as métricas OO são coletadas a partir do bytecode Java. Na Seção 4 é apresentado o estudo de caso onde as métricas são aplicadas sobre dois sistemas fazendo-se a avaliação da co-relação entre os valores das métricas e a propensão a falhas de cada classe dos sistemas. A Seção 5 apresenta os comentários finais sobre o trabalho.

2 Trabalhos Relacionados

Métricas de software existem de longa data. Várias delas foram propostas e utilizadas para medir características de programas "tradicionais", dentro do paradigma estruturado. É o caso, por exemplo, de medidas como Linhas de Código e complexidade Ciclomática [20]. Com o surgimento do paradigma orientado a objetos (OO) foram criadas novas métricas que abordassem as particularidades dos novos conceitos introduzidos com esta tecnologia, tais como classe, polimorfismo e herança.

As métricas para sistemas OO têm como função a obtenção das características numéricas das estruturas internas do software, que refletem a complexidade de cada componente individual, como métodos e classes; e na complexidade externa, que mede as interações entre as entidades, tais como acoplamento e herança [6]. Dentre as métricas OO propostas destacam-se dois grupos, conhecidos como as métricas CK (Chidamber e Kemerer) e as métricas LK (Lorenz e Kidd).

Chidamber e Kemerer proporam em [7] um conjunto de seis métricas para melhorar o processo de desenvolvimento de projetos OO. Foram coletadas amostras empíricas destas métricas em 2 projetos comerciais diferentes para demonstrar como as mesmas poderiam ser utilizadas para melhorar o processo. A tabela 1 descreve as 6 métricas de projeto OO propostas por CK, de forma sucinta.

Tabela 1: Descrição Sucinta as Métricas CK

Métrica	**Descrição**
NOC	Número de Filhos
DIT	Profundidade da Árvore de Herança
WMC	Número ponderado de métodos por classe
LCOM	Falta de coesão entre os métodos
RFC	Resposta para uma classe
CBO	Acoplamento entre objetos

Lorenz e Kidd proporam em [15] 11 métricas de projeto (*project metrics*) e 32 métricas de desenho (*design metrics*). As métricas de projeto medem aspectos dinâmicos do sistema em um nível mais alto de abstração, para predizer esforço. Já as métricas de desenho medem aspectos estáticos do software buscando medir a qualidade do software que tem sido desenvolvido [15]. A tabela 2 descreve as principais métricas de desenho OO propostas por LK, de forma sucinta.

Tabela 2: Descrição Sucinta das Métricas LK

Métrica	**Descrição**
NPIM	Número de métodos de instância públicas na classe
NIV	Número de variáveis de instância na classe
NCM	Número de métodos classe na classe
ANPM	Número médio de parâmetros por método
UMI	Uso de herança múltipla
AMZ	Tamanho médio do método
NMOS	Número de métodos sobrescritos na subclasse
NMIS	Número de métodos herdados pela subclasse
NMAS	Número de métodos adicionados pela subclasse

Vários trabalhos têm sido desenvolvidos na tentativa de validar as métricas empiricamente no seu relacionamento com propensão à ocorrência de falhas nos módulos do software. Em [1], os autores investigaram a propensão de uma classe conter falhas a partir dos dados coletados pelas seis métricas de CK no código fonte. Para validar estas métricas

foram utilizados dados de desenvolvimento de oito projetos em C++ de médio porte. Os autores fizeram uma análise quantitativa e empírica dos dados coletados. Os resultados mostraram que as métricas NOC, RFC e DIT estão significantemente relacionadas com a propensão de se encontrar falha, a métrica CBO está medianamente relacionada com tendência à falha e a métrica WMC está pouco relacionada com a propensão de se encontrar falha. Em todos os casos a métrica LCOM se mostrou insignificante para indicar uma classe como sendo propensa a conter falha. Além disso, os autores afirmam que as métricas são melhores indicadores que as métricas tradicionais uma vez que podem ser aplicadas nas fases iniciais do ciclo de vida do software.

Em [3] foram explorados empiricamente os relacionamentos existentes entre métricas OO e a probabilidade de detecção de falha nas classes do sistema durante a atividade de teste. Foram utilizados sistemas C++ e calculadas várias métricas de acoplamento, de coesão, de herança e de tamanho. A análise dos dados coletados para cada medida foi descrito em quatro estágios: análise de distribuição de dados, análise de componente principal, construção do modelo de predição (regressão logística simples) e correlação com o tamanho. Os autores afirmaram que:

- Muitas das medidas de acoplamento, coesão e herança capturam dimensões parecidas dos dados e isso ocorre pelo fato de que muitas medidas propostas na literatura são baseadas em idéias e hipóteses comparáveis, sendo assim redundantes. Por isso, o resultado da avaliação para essas métricas apresenta dados redundantes;

- Pela análise simples, muitas das medidas de herança e acoplamento são fortemente relacionadas à probabilidade de detecção de falha em uma classe, em particular, acoplamento através de invocações de método (CBO) e a profundidade de uma classe na hierarquia de herança (DIT). Medidas de coesão não se mostraram significantes pela dificuldade de se medir o conceito e do fraco entendimento a que este atributo se propõe capturar;

- Pela análise múltipla, através de algumas das medidas de herança e acoplamento, pode-se predizer quais classes possuem a maioria das falhas.

Em [5], a investigação empírica foi sobre um sistema de um produto de telecomunicações OO com aproximadamente 133.000 linhas de C++. As métricas de CK foram utilizadas para, dentre outros, considerar a sua efetividade e usabilidade em termos de facilidade para coleta de dados e utilidade para predizer falhas e identificar fatores associados a altos níveis de falhas. Foram coletadas: variáveis na análise e no projeto que caracterizam a estrutura do sistema, variáveis gerenciais e dados de estatísticas descritivas. Sistemas de predição foram construídos e como resultados, dentre outros, os autores verificaram pouco uso de herança e polimorfismo e encontraram a métrica DIT relacionada à ocorrência de falhas;

Em [9], é investigado se o tamanho da classe tem um efeito de distorcer os valores das métricas, o que os autores definem como *confused effect*. Os autores descreveram uma metodologia empírica detalhada para identificar estes efeitos e fizeram um estudo de caso aplicando as métricas CK e um subconjunto das métricas de LK em um sistema

de telecomunicações escrito em C++. Para avaliar os resultados, a técnica de regressão logística foi utilizada. Dentre os resultados obtidos, os autores verificaram, pela análise simples, que as métricas WMC, CBO, RFC e NMAS estão associadas com propensão de se encontrar falha.

Outros trabalhos foram feitos investigando-se o relacionamento entre as métricas CK e propensão de se encontrar falha. Os resultados variam para algumas métricas, mas para as métricas RFC e WMC a associação com a propensão de se encontrar falha é encontrada em todos os trabalhos da área. Em [4], apenas a métrica NOC não foi associada com a propensão de se encontrar falha (a métrica WMC não foi utilizada). Em [2], as métricas NOC e CBO e em [18], as métricas CBO e DIT não foram associadas à propensão de se encontrar falha.

3 Coleta de métricas em bytecode Java

Como destacado anteriormente, as métricas OO podem ser coletadas a partir do projeto de classes ou do código fonte das mesmas. Existem situações entretanto, em que tais informações podem ser úteis ao engenheiro de software e nas quais não se dispõe do código fonte e menos ainda do projeto das classes. É o caso, por exemplo, do teste de programas baseados em componentes, em que parte do código pode ser produzido por terceiros e da qual não se tem acesso ao código fonte. É o caso também de re-engenharia a partir do código objeto. A medida de complexidade das classes pode dar uma indicação do esforço necessário nessas atividades.

Programas objeto Java, ou mais precisamente, programas representados através de bytecode que possa ser executado pela Máquina Virtual Java [14], mantêm muito das informações relativas às características de orientação a objetos que são utilizadas nas métricas OO e que normalmente são obtidas no código fonte. Por exemplo, da descrição de uma determinada classe, no formato definido para a JVM pode-se extrair:

- O nome da sua super-classe;
- O tipo e assinatura de seus métodos;
- O tipo e o nome das suas variáveis de instância e de classe;
- Os métodos de outras classes que são invocados;
- A relação de classes das quais esta classe depende;

A ferramenta JaBUTi [19] foi projetada com o principal objetivo de permitir que sejam aplicados critérios de teste estrutural diretamente em programas objeto Java. Ela faz a análise de arquivos de classe Java e deriva diversas informações, entre elas:

- A estrutura hierárquica do programa sob análise. A partir de uma determinada classe "raiz", a ferramenta descobre quais são as demais classes e interfaces requeridas e constrói a estrutura hierárquica. O testador pode escolher quais pacotes devem

ser considerados dentro do escopo do program, excluindo, por exemplo, classes localizadas em bibliotecas de terceiros. Como padrão, assume-se que as classes de sistema (da API Java) estão fora do escopo do programa. A Figura 1 mostra um exemplo.

- Para cada classe dentro do escopo do programa, quais são os métodos definidos; e
- Para cada método nas classes dentro do escopo do programa, o grafo de fluxo de controle e informação sobre definições e usos de variáveis dentro do método.

Com essas informações é possível através da instrumentação diretamente do bytecode, executar casos de teste e colher informação sobre a cobertura destes em relação ao programa sob teste. É possível também colher-se informação sobre muitas das métricas OO propostas na literatura, conforme será descrito adiante.

Um ponto importante a se notar é que a escolha da estrutura de programa comentada acima, na qual define-se um conjunto de classes que fazem parte do escopo do programa sob análise, ignorando as classes que não são de interesse, tem influência em algumas das métricas implementadas, a saber, naquelas métricas relacionadas com a hierarquia de classes. Por exemplo, na Figura 1 as classes *Graph* e *GraphNode* têm ambas valor 1 para a métrica DIT, independentemente do número de subclasses que possuam fora do escopo do programa.

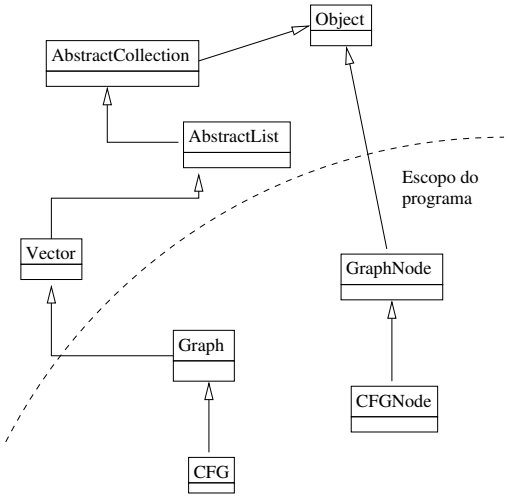

Figura 1: Exemplo da estrutura de um programa na ferramenta JaBUTi.

Utilizando a estrutura do programa e as informações coletadas para cada classe e cada método, foram implementadas 27 métricas na ferramenta JaBUTi. Das métricas implementadas: 10 são métricas de LK; 4 são variações das métricas LK, que aproveitam o fato de serem aplicadas no bytecode e não no programa fonte; 6 são métricas de CK; 5 são variações das métricas CK e 2 são variações da métrica tradicional Complexidade Ciclomática de McCabe (CC). Segue uma descrição sucinta de cada uma delas.

Métricas de LK

- Número de métodos de instância públicas na classe - *Number of public instance methods in a class* - NPIM: métrica calculada através da contagem do número de métodos de instância públicos na classe;

- Número de variáveis de instância na classe - *Number of instance variables in a class* - NIV: métrica calculada através da contagem do número de variáveis de instância na classe, o que inclui as variáveis *public*, *private* e *protected*;

- Número de métodos de classe na classe - *Number of class methods in a class* - NCM: métrica calculada através da contagem do número de métodos *static* na classe;

  Variação (NCM2): métrica calculada através da contagem do número de métodos *public static* na classe;

- Número de variáveis de classe na classe - *Number of class variables in a class* - NCV Métrica calculada através da contagem do número de variáveis *static* na classe.

- Número médio de parâmetros por método - *Average number of Parameters per method* - ANPM: métrica calculada através da divisão entre o somatório do número de parâmetros de cada método da classe pelo número total de métodos da classe;

  Variação (MNPM): número máximo de parâmetros nos métodos da classe;

- Número de interfaces implementadas - *Number of Interfaces Implemented* - NII: Variação da métrica UMI (Uso de herança múltipla - *Use of multiple inheritance*) já que a linguagem não permite herança múltipla.

- Tamanho médio do método - *Average method size* - AMZ_LOCM: métrica calculada através da divisão entre a soma do número de linhas de código dos métodos da classe pelo número de métodos na classe (soma dos métodos de instância e classe). Note-se que esta métrica relaciona-se realmente ao número de linhas de código no programa fonte, uma vez que do bytecode é possível extrair este valor;

  Variação (AMZ_SIZE): semelhante a AMZ_LOCM mas utiliza o número de instruções do bytecode como tamanho do método e não o número de linhas de código do programa fonte.

- Número de métodos sobrescritos na subclasse - *Number of methods overridden by a subclass* - NMOS: métrica calculada através da contagem do número de métodos definidos na subclasse com a mesma assinatura de um método em uma das suas superclasses (diretas ou indiretas). Nesta e nas demais métricas que requerem um passeio nas superclasses de uma classe somente as classes no escopo do programa são utilizados na busca. Além disso, construtores (que no program objeto recebe sempre o nome <*init*>) e inicializadores estáticos (<*cinit*>) não são contados;

- Número de métodos herdados pela subclasse - *Number of Methods inherited by a subclass* - NMIS: métrica calculada através da contagem do número de métodos herdados pela subclasse de suas superclasses;

- Número de métodos adicionados pela subclasse - *Number of Methods added by a subclass* - NMAS: métrica calculada através da contagem do número de novos métodos adicionados pela classe;

- Índice de Especialização - *Specialization index* - SI: métrica calculada através da divisão entre o resultado da multiplicação de NMOS e DIT (métrica de CK) pelo número total de métodos.

Métricas de CK

- Número de Filhos - *Number of Children* - NOC: métrica calculada através da contagem do número de subclasses imediatas subordinadas à classe na árvore de hierarquia de herança;

- Profundidade da Árvore de Herança - *Depth of Inheritance Tree* - DIT: é o maior caminho da classe à raiz na árvore de hierarquia de herança. Interfaces também são consideradas, ou seja, o caminho através de uma hierarquia de interfaces também pode ser o que dá a profundidade de uma classe. Como a representação do programa utilizada não inclui todas as classes até a raiz da árvore de hierarquia, é utilizado o caminho da classe até a primeira classe que não pertence à estrutura do programa;

- Número ponderado de métodos por classe - *Weighted Methods per Class* - WMC: métrica calculada através da soma da complexidade de cada método. Na proposição original da métrica não se define qual tipo de complexidade pode ser utilizada, assim são definidas as seguintes variações:

  WMC_1: valor 1 como complexidade de cada método. Número de métodos na classe;

  WMC_CC: métrica Complexidade Ciclomática (CC) para a complexidade de cada método. Soma-se o valor CC de cada método da classe;

  WMC_LOCM: métrica Linhas de Código (LOCM) para a complexidade de cada método. Soma-se o valor LOCM de cada método da classe;

  WMC_SIZE: tamanho do método para a complexidade de cada método.

- Falta de Coesão entre os métodos - *Lack of Cohesion in Methods* - LCOM: métrica calculada através da contagem do número de pares de métodos na classe que não compartilham variáveis de instância menos o número de pares de métodos que compartilham variáveis de instância. Quando o resultado é negativo, a métrica tem o valor zero. Os métodos estáticos não são considerados na contagem, uma vez que só as variáveis de instância são tomadas. São aplicadas as seguintes variações:

  LCOM_2: considera somente a coesão entre métodos estáticos, considerando-se obviamente o compartilhamento de variáveis estáticas;

  LCOM_3: considera a coesão de métodos estáticos ou de instância. Calculada pela soma de LCOM com LCOM_2.

- Resposta para uma classe - *Response for a Class* - RFC: métrica calculada através da soma do número de métodos da classe mais os métodos que são invocados diretamente por eles. É o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto de uma classe ou por algum método da classe. Quando um método polimórfico é chamado para diferentes classes, cada diferente chamada identificada no bytecode é contada uma vez;

- Acoplamento entre objetos - *Coupling Between Object* - CBO: Há acoplamento entre duas classes quando uma classe usa métodos e/ou variáveis de instância de outra classe. Métrica calculada através da contagem do número de classes às quais uma classe está acoplada de alguma forma. O valor CBO de uma classe A é o número de classes das quais a classe A utiliza algum método e/ou variável de instância, o que inclui o acoplamento baseado em herança, visto que um construtor da superclasse é sempre chamado no construtor de A.

Outras Métricas

- Complexidade Ciclomática de McCabe - *Cyclomatic Complexity Metric* - CC: métrica que calcula a complexidade do método, através dos grafos de fluxo de controle que descreve a estrutura lógica do método. Os grafos de fluxo consistem de nós e ramos, onde os nós representam comandos ou expressões e os ramos representam a transferência de controle entre estes nós. A métrica pode ser calculada de várias formas, sendo uma delas: número de ramos - número de nós + 2. Mais esclarecimentos sobre a métrica podem ser vistos em [17] e [20]. São aplicadas as seguintes variações para a métrica, além de WMC_CC vista acima:

  CC_AVG: valor médio dos valores da métrica CC de cada método da classe;

  CC_MAX: valor máximo obtido dos valores da métrica CC entre os métodos da classe.

Na próxima seção é apresentado um estudo de caso, onde a ferramenta JaBUTi é utilizada para coletar estas métricas a partir do código objeto de dois sistemas desenvolvidos em Java. Embora esta coleta tenha sido utilizada para tentar relacionar as métricas com a propensão a falhas das classes analisadas, seu objetivo maior não é o de avaliar ou validar as métricas em si, mas apenas o de mostrar a factibilidade de coletá-las a partir do bytecode, ao invés de a partir do código fonte.

4 Estudo de Casos

As métricas implementadas na ferramenta JaBUTi, descritas na seção anterior, foram utilizadas num estudo de caso utilizando dois sistemas. O primeiro, chamado μCode [16], considerado um sistema mais simples e o segundo, a própria ferramenta JaBUTi, um sistema um pouco maior e mais complexo.

O estudo de caso procurou relacionar as métricas OO com a propensão a falhas, a exemplo do que foi feito em trabalhos encontrados na literatura como [1–5, 18]. Para tal,

foram utilizadas duas versões de cada sistema, sendo uma a evolução da outra. Sobre a primeira versão (mais antiga) foram aplicadas as métricas OO através da ferramenta JaBUTi. A segunda versão (atualizada) serviu como referência para que se pudesse avaliar em quais classes foram encontrados mais falhas. Foram consideradas "falhas" na versão original, todas as classes que sofreram alguma modificação de código na versão atualizada. Essa abordagem não é exata, pois uma modificação numa classe não indica necessariamente a existência de uma falha. Porém, com a inexistência de sistemas, dos quais se consiga um registro de erros, decidiu-se por adotar essa forma para avaliar as classes com propensão a falhas.

Fez-se então a análise dos dados procurando relacionar o valor das métricas coletadas com a quantidade de modificação que a classe sofreu. Note-se que o objetivo principal desse estudo de caso não foi realmente avaliar as métricas OO em relação a sua capacidade em indicar classes propensas à existência de falhas, mas sim mostrar a factibilidade que tais experimentos, como os realizados por [1–5, 18] possam ser realizados no nível de código objeto Java.

A técnica estatística utilizada para as análises dos dados foi a regressão logística. Esta técnica baseia-se na construção de um modelo linear que é usado para explorar o relacionamento (se existir) entre uma variável resposta binária (variável dependente) e uma ou mais variáveis independentes [8]. Neste trabalho, a variável dependente é a ocorrência de falha na classe e as variáveis independentes são os valores das métricas na classe. Para a construção do modelo é necessário estimarem-se os seus coeficientes. O método de estimativa de coeficientes utilizado foi o método de máxima verosimilhança. Mais esclarecimentos desta técnica utilizada nos trabalhos de validação empírica das métricas orientadas a objetos podem ser obtidos em [8] e [13].

Da mesma forma que os trabalhos anteriormente executados na área descritos na Seção 2 [1–5, 18], foram construídos modelos de regressão logística simples (apenas uma variável independente) para cada métrica, através dos quais foram selecionadas as métricas que fizeram parte do modelo de regressão logística múltiplo. Para cada modelo de regressão logística simples construído foram obtidos:

- Testes estatísticos Wald Chi-Square, Score e teste da máxima verosimilhança (*Likelihood Ratio* - G) e seus respectivos *p-value* associados para testar a hipótese nula de que todos os coeficientes de regressão são iguais a 0;
- Estimativas de máxima verosimilhança (coeficiente estimado, erro padrão, estatística Wald Chi-Square e seu *p-value* associado);
- Estimativa da razão de chance (*odds ratio*).

As variáveis selecionadas para estarem presentes no modelo de regressão logístico múltiplo foram as que apresentaram valores altos para as estatísticas G e Wald Chi-Square e nível de significância de 0.25 nos *p-value* associados a estas estatísticas. Nos casos em que, se resultar um número elevado de variáveis selecionadas, foi executado o método de seleção de variáveis *Best Selection*.

A seguir, serão descritos os resultados obtidos das análises estatísticas das métricas aplicadas aos dois sistemas.

4.1 μCode

A ferramenta μCode [16] é uma ferramenta freeware disponível na Internet no site: http://mucode.sourceforge.net. Ela é uma API Java que dá suporte ao desenvolvimento de programas com código móvel. Ela é relativamente pequena e possui 16 classes e 4 versões liberadas. Para análise dos dados, foram utilizadas a versão (1.0), liberada em 7/9/2000 para aplicação das métricas e a versão atual (1.03), liberada em 1/8/2002. Das 16 classes que compõem o programa, 6 sofreram algum tipo de alteração da versão inicial para a versão atualizada.

Foram construídos modelos simples para cada uma das 27 métricas coletadas e um modelo múltiplo para as métricas resultantes da análise simples, conforme descrito anteriormente. Os resultados mais relevantes obtidos destes modelos simples e do modelo múltiplo final, podem ser vistos nas tabelas 3 e 4, respectivamente.

Tabela 3: Resultados relevantes da análise simples no μCode

Métrica	**Coef. Estim.**	**Erro Padrão**	**Wald Chi-Square**	**p-value**	$\Delta\Psi$
NCV	-0.215	0.1865	1.3287	0.249	0.807
NIV	0.9486	0.6214	2.3306	0.1269	1.880
NII	1.2953	0.8095	2.5604	0.1096	3.652
LCOM_3	0.0106	0.00763	1.9458	0.1630	1.011
RFC	0.0173	0.0115	2.2519	0.1334	1.017
WMC_SIZE	0.0026	0.00178	2.1132	0.1460	1.003
WMC	0.1489	0.0803	3.4352	0.0638	1.161
NPIM	0.2888	0.1550	3.4713	0.0624	1.323
CBO	0.1769	0.0977	3.2818	0.0701	1.194
WMC_LOCM	0.0108	0.0077	1.9369	0.1640	1.011

Tabela 4: Resultado final da análise múltipla no μCode

Métrica	**Coef. Estim.**	*Standard Error*	**Wald Chi-Square**	**p-value**	*odds ratio*
Intercept	-2.0112	0.9531	4.4531	0.0348	
NPIM	0.2798	0.1487	3.5403	0.0599	1.323

Alguns dos resultados da análise dos dados:

- Na primeira etapa da análise dos dados (análise simples), 17 das 27 métricas calculadas foram descartadas e consideradas insignificantes no relacionamento com a propensão de encontrar falhas. Na segunda etapa (análise múltipla), após ter sido criado um modelo com as 10 variáveis consideradas significantes, os resultados foram insatisfatórios, pelo fato de todas as métricas apresentarem altos níveis de significância para o *p-value* associado à estatística Wald Chi-Square. Desta forma, foi executado o método de seleção de variáveis *Best Selection* e após a construção de alguns modelos para encontrar-se o modelo mais ajustado, chegou-se no modelo com apenas a métrica NPIM;

- Os baixos valores das métricas de herança DIT, NOC, NMIS, NMAS, NMOS, NII e SI indicam que neste sistema houve um uso insignificante de herança;

Em relação ao resultado apresentado por outros trabalhos semelhantes, pode-se notar:

- Como em [1], a métrica RFC foi encontrada relacionando-se com a propensão de encontrar falhas, assim como CBO e WMC, mas com a diferença de que a métrica LCOM não se mostrou insignificante em todos os casos;
- Em [5], os valores de NOC e DIT se mostraram com valores mínimos, o mesmo ocorrido com este estudo de caso;
- Como em [9], as métricas RFC, CBO, WMC foram encontradas pela análise simples associadas com propensão de encontrar falhas;
- Como em [4] e [2], a métrica NOC e em [18] a métrica DIT não foram associadas com propensão a falha;

4.2 JaBUTi

O segundo programa no qual as métricas foram aplicadas é a própria ferramenta JaBUTi. Ela possui 50 classes e algumas versões de desenvolvimento. Para análise dos dados, da mesma forma foram utilizadas duas versões, a primeira de 6/11/2002 e a segunda de 17/01/2003. Destas 50 classes, 13 sofreram algum tipo de alteração.

Os resultados mais relevantes obtidos destes modelos simples e do modelo múltiplo final, podem ser vistos nas tabelas 5 e 6, respectivamente.

Tabela 5: Resultados relevantes da análise simples na ferramenta JaBUTi

Métrica	**Coef. Estim.**	**Erro Padrão**	**Wald Chi-Square**	**p-value**	$\Delta\Psi$
NMOS	0.799	0.6344	1.5863	0.2079	2.223
CC_MAX	0.0564	0.0301	3.5061	0.0611	1.058
ANPM	-0.6508	0.4845	1.8044	0.1792	0.522
NCM_2	0.4576	0.278	2.7095	0.0998	1.58
NCM	0.2361	0.1691	1.9503	0.1626	1.266
NMAS	0.4757	0.2286	4.3284	0.0375	1.609
LCOM	0.012	0.00791	2.3054	0.1289	1.012
NII	1.3863	0.7458	3.4549	0.0631	2.844
LCOM_3	0.0165	0.00981	2.8445	0.0917	1.017
AMZ_LOCM	0.0765	0.0427	3.2046	0.0734	1.079
NOC	1.4329	0.7043	4.1391	0.0419	4.191
RFC	0.0179	0.00641	7.7728	0.0053	1.018
WMC	0.0622	0.0333	3.4926	0.0616	1.064
NPIM	0.0786	0.0446	3.1085	0.0779	1.082
CBO	0.0414	0.0283	2.1475	0.1428	1.042
WMC_CC	0.0192	0.0104	3.4014	0.0651	1.019
CC_AVG	0.4993	0.1962	6.4774	0.0109	1.648
NMIS	0.0635	0.0333	3.6252	0.0569	1.066

Alguns dos resultados da análise dos dados:

- Na primeira etapa da análise dos dados (análise simples), 9 métricas foram descartadas e consideradas insignificantes no relacionamento com a propensão de encontrar falhas. Na segunda etapa (análise múltipla), foi criado um modelo com as 18 variáveis consideradas significantes, porém os resultados foram insatisfatórios. Da mesma forma, foi executado o método *Best Selection* e após a construção de alguns modelos para encontrar-se o modelo mais ajustado, chegou-se no modelo com as métricas RFC, NOC, CC_AVG e ANPM;
- Os baixos valores das métricas SI, NII, NMOS, DIT e NOC indicam que neste sistema também houve um uso insignificante de herança. Para estas métricas de herança foram considerados valores baixos, valores até 2.

Tabela 6: Resultado final da análise múltipla na ferramenta JaBUTi

Métrica	**Coef. Estim.**	**Erro Padrão**	**Wald Chi-Square**	**p-value**	$\Delta\Psi$
Intercept	-2.9856	0.9834	9.2170	0.0024	
RFC	0.0200	0.00955	4.3988	0.0360	1.020
CC_AVG	0.9666	0.3864	6.2579	0.0124	2.629
NOC	2.0884	1.0420	4.0168	0.0451	8.072
ANPM	-2.9358	1.1361	6.6775	0.0098	0.053

Na comparação com outros trabalhos pode-se notar que:

- Assim como em [1], as métricas NOC e RFC foram encontradas relacionadas à propensão de encontrar falhas, porém, neste estudo de caso a métrica LCOM não se mostrou insignificante em todos os casos;
- Em [5], os valores de NOC e DIT se mostraram com valores mínimos, o mesmo ocorrido com este estudo de caso;
- Da mesma forma que em [9], pela análise simples, as métricas RFC, CBO, WMC e NMAS também se mostraram associadas com propensão de encontrar falhas;
- Como em [3], na análise simples, as métricas de herança e acoplamento também se mostraram fortemente relacionadas à probabilidade de se encontrar falhas, mas, as métricas de coesão não se mostraram insignificantes;
- A métrica DIT mostrou-se insignificante na associação com propensão a falha, como em [2].

5 Conclusões

Atualmente, com a necessidade de obterem-se produtos de software de melhor qualidade e de se melhorar o processo de desenvolvimento, as métricas ganharam uma atenção diferenciada. Com a utilização do paradigma OO e sem a possibilidade de se aproveitar as métricas do paradigma estruturado, que não consideram os novos conceitos introduzidos com esta nova tecnologia, tais como classe, polimorfismo, herança e encapsulamento, algumas métricas OO têm sido propostas.

Muitas destas métricas são aplicadas diretamente no nível de código, requerendo para isso, a disponibilidade do código fonte. Existem casos, porém, em que pode-se fazer uso destas métricas sem que se tenha acesso ao código fonte. Para tais situações, uma alternativa é fazer a coleta das métricas diretamente no código objeto. No caso específico de programas Java, tal abordagem torna-se possível, dada à estrutura do código objeto (bytecode) Java que mantém muito das informações e características de orientação a objetos, definidas no código fonte.

Este trabalho propôs a aplicação das mais importantes métricas OO que têm sido definidas na literatura de orientação a objetos em uma estrutura de programa gerada a partir do código objeto Java. Em um estudo de caso, repetiu-se o trabalho de avaliação das métricas realizado por [1]. O objetivo principal não foi validar ou analisar as métricas em si, mas mostrar que o mesmo estudo realizado no nível de código fonte pode ser repetido no nível de código objeto.

Os resultados gerados a partir dos ambientes dos estudos de casos descritos neste artigo foram similares aos descritos na literatura para esta área, com a identificação de métricas, como RFC, CBO e WMC, relacionadas à propensão de encontrar falhas nos módulos de software, em conjunto com outras métricas de LK. Como observado na seção 2, os baixos valores para as métricas DIT e NOC demonstraram pouco uso de herança nos sistemas utilizados como estudos de casos, o qual também verificou-se com este. Porém, os resultados diferiram em alguns aspectos, devido às características peculiares de cada ambiente nos quais as métricas foram aplicadas. Ao observar que outros trabalhos relacionados à área apontam para resultados diferentes, quanto a aplicação de métricas no código fonte, aliada à compatibilidade de alguns resultados deste trabalho com aqueles, deduz-se que a aplicação de métricas no código objeto não acarreta em significantes implicações em relação ao código fonte, tornando-se, desta forma, uma ferramenta útil para o manejo do testador, em casos de indisponibilidade do código fonte.

Referências

[1] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions On Software Engineering*, 22(10):751–761, Outubro 1996.

[2] A. Binkley and S. Schach. Validation of the coupling dependency metric as a predictor of run-time fauilures and maintenance measures. In *Proc. 20th Int'l Conf. Software Engineering*, pages 452–455, 1998.

[3] L. Briand, J. Wüst, J. W. Daly, and V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal Systems and Software*, 51:245–273, 2000.

[4] L. Briand, J. Wüst, S. Ikonomovski, and H. Lounis. A comprehensive investigation of quality factors in object-oriented designs: An industrial case study. Technical Report ISERN-98-29, Int'l Software Eng. Research Network, 1998.

[5] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions On Software Engineering*, ?

[6] L. B. Chaves. *Uma Avaliação Empírica de Métricas para Programas Orientados a Objeto no contexto de Teste de Software.* PhD thesis, Universidade Federal do Paraná, Curitiba, 2001.

[7] S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions On Software Engineering*, pages 467–493, 1994.

[8] D. Collet. *Modelling Binary Data.* Chapman & Hall, 1994.

[9] K. Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions On Software Engineering*, 27(7), Julho 2001.

[10] R. Gunning. *Techniques of Clear Writing.* Nova York: McGraw-Hill, 1962.

[11] W. Harrison. Using software metrics to allocate testing resources. *J. Management Information Systems*, 4(4):93–105, 1988.

[12] W. Harrison. Software measurement: A decision-process approach. *Advances in Computers*, 39:51–105, 1994.

[13] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression.* Wiley-Interscience, 1989.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, second edition, 1999.

[15] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics - A Practical Guide.* Prentice Hall, 1994.

[16] G. P. Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. 2nd International Workshop on Mobile Agents 98 (MA'98)*, pages 160–171, Stuttgart (Germany), September 1998.

[17] R. S. Pressman. *Engenharia de Software.* Makron Books, 4ł edition, 1997.

[18] M-H Tang, M-H Kao, and M-H Chen. An empirical study on object oriented metrics. In *Proc. Sixth Int'l Software Metrics Symp.*, pages 242–249, 1999.

[19] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and E. W. Wong. JaBA: A Java Bytecode Analyzer. In *Proc. Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas)*, Gramado - RS, Outubro 2002.

[20] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric, 1996.