Uma Ferramenta para Auxílio na Avaliação de Usabilidade de APIs

David L. M. Bentolila, Cleidson R. B. de Souza

Faculdade de Computação – Universidade Federal do Pará (UFPA)
Belém – PA – Brasil

davidbentolila@gmail.com, cdesouza@ufpa.br

Abstract. *The use of Application Programming Interfaces (APIs) nowadays is obviously very broad. They are used in various systems, ubiquitous or not, through the libraries use. These libraries are used for many reasons, mainly to code reuse, which can result in system development time reduction. This paper describes the MetriX tool, which aims to support users in APIs' usability evaluation tasks through metrics of complexity. The tool uses software complexity metrics for creating views that will support user during the evaluation.*

Resumo. *A utilização de Application Programming Interfaces (APIs) nos dias atuais é visivelmente muito ampla. Elas são utilizadas nos mais diversos sistemas, ubíquos ou não, através do uso de bibliotecas. Estas bibliotecas são utilizadas por diversas razões, principalmente visando a reutilização de código, o que pode implicar em diminuição no tempo de desenvolvimento de um sistema. Este trabalho descreve a ferramenta MetriX, que tem como objetivo auxiliar usuários em tarefas de avaliação de usabilidade de APIs através de métricas de complexidade. A ferramenta utiliza métricas de complexidade de software para gerar visualizações que irão auxiliar o usuário durante a avaliação de APIs e a escolha de qual API utilizar baseado na facilidade de uso da mesma.*

1. Introdução

A visão inovadora de Weiser [Weiser 1991] da computação ubíqua somente se tornou possível recentemente com a queda dos custos dos processadores, redução do tamanho dos circuitos integrados, entre outros fatores. A perspectiva da computação ubíqua é que computadores se tornarão cada vez mais presentes no cotidiano das pessoas. Este novo contexto requer "novos métodos, ambientes, técnicas, modelos, dispositivos, padrões de arquitetura e de projeto capazes de auxiliar projetistas de sistemas de hardware e software" [Carvalho, Brayner et al. 2006]. Por isto, facilitar o projeto de sistemas ubíquos é de extrema importância. De fato, este é o quinto dos grandes desafios propostos pela Sociedade Brasileira de Computação.

Em sistemas ubíquos, e até mesmo sistemas "convencionais", os componentes de software precisam se comunicar para trocar informações (ou dados), visto que tais sistemas são construídos a partir de partes menores, chamadas módulos ou componentes, para lidar com a complexidade dos mesmos [Simon 1996]. Esta troca de informações comumente é realizada através de APIs (*application programming interfaces*). Segundo o Software

Engineering Institute (SEI)[1], uma API é uma tecnologia que facilita a troca de mensagens ou dados entre duas ou mais aplicações. Inicialmente, APIs eram tecnologias baseadas apenas em simples chamadas de sub-rotinas. Entretanto, elas foram expandidas e hoje incluem novas funcionalidades e permitem interoperabilidade e mudanças em sistemas, permitindo compartilhamento de dados entre diversas aplicações.

Dada a importância de APIs no projeto de sistemas computacionais, fazer um bom projeto de APIs torna-se cada vez mais importante. Um dos aspectos do projeto de uma API é a sua usabilidade [Ellis 2007], isto é, quão fácil ou difícil é utilizar esta API. Este trabalho pretende apontar onde desenvolvedores poderão encontrar possíveis problemas de usabilidade de maneira *automática*. Neste trabalho a usabilidade de uma API esta diretamente relacionada com sua complexidade, ou seja, quanto mais complexa uma API, mais difícil será utilizar a mesma. A complexidade dos componentes de uma API pode influenciar diretamente no custo de manutenção de uma API [Bandi et al. 2003], por isto, a análise prévia da complexidade tem grande importância, podendo baratear o custo da manutenção, antes mesmo de uma API tornar-se disponível. A complexidade de uma API é calculada a partir da utilização de métricas de software. Para ser mais específico, as métricas de Bandi [Bandi et al. 2003] – Tamanho da Interface, Nível de Interação e Complexidade do Argumento da Operação – são utilizadas para calcular a complexidade das APIs. Métricas são boas para resumir aspectos particulares de coisas e para detectar pontos importantes em grande quantidade de dados [Lanza 2005]. Elas são um bom mecanismo para sintetizar muitos detalhes de um software, entretanto a apresentação das mesmas pode ser problemática, uma vez que pequenos detalhes podem passar despercebidos em meio a uma enorme quantidade de dados. A melhor forma encontrada para solucionar este problema foi através da utilização de técnicas de visualização de informação.

Em resumo, a abordagem proposta neste trabalho permite a avaliação *automática* da usabilidade de APIs através de métricas de complexidade de APIs e técnicas de visualização de informação. Estas abordagens são combinadas em uma ferramenta chamada Metrix. A ferramenta proposta neste trabalho gera visualizações TreeMap [Bederson 2002], StarPlot [Júnior 2008] e grafos de dependência baseadas em dados obtidos de APIs Orientadas a Objetos. Estas visualizações fornecem informações que podem ser usadas por usuários ou por desenvolvedores de uma API. Por exemplo, os usuários poderão definir qual API é mais simples ou mais complexa de usar, já o gerente de projeto de uma API poderá alocar um determinado número de pessoas para uma tarefa de manutenção, baseado na complexidade dos módulos onde a manutenção será realizada.

O restante deste trabalho está dividido da seguinte maneira. A seção 2 apresenta alguns conceitos sobre APIs, sua definição, aspectos de sua usabilidade e vantagens de seu uso. Em seguida, a seção 3 define complexidade, sua relação com a usabilidade e conceitua brevemente métricas de software, assim como apresenta as métricas propostas por Bandi [Bandi et al. 2003] e por Boxall e Araban [Boxall e Araban 2004]. A seção 4 apresenta a ferramenta proposta neste trabalho, e as visualizações utilizadas pela ferramenta. A seção 5 apresenta algumas limitações da abordagem utilizada na ferramenta. Já a seção 6 apresenta alguns trabalhos relacionados. Por fim a seção 7 apresenta as considerações finais deste trabalho.

---

[1] http://www.sei.cmu.edu/str/descriptions/api.html

2. Application Programing Interfaces – API

O princípio de ocultamento de informação proposto por [Parnas 1972] afirma que em uma aplicação deve-se separar a implementação de sua especificação. Além disso, este princípio recomenda que a implementação esteja sempre oculta. Este princípio é um dos pilares da engenharia de software e foi implementado em linguagens orientadas a objetos através de técnicas como encapsulamento, polimorfismo e separação das especificações de suas implementações [Larman 2001]. Segundo o Software Engineering Institute (SEI), uma API é uma tecnologia que facilita a troca de mensagens ou dados entre duas ou mais aplicações. Inicialmente, APIs eram tecnologias baseadas apenas em simples chamadas de sub-rotinas. Entretanto, elas foram expandidas e hoje incluem novas funcionalidades e permitem interoperabilidade e mudanças em sistemas, permitindo compartilhamento de dados entre diversas aplicações.

De maneira simplificada, uma API orientada a objetos corresponde a um conjunto de especificações de interfaces, métodos e classes. Neste trabalho, será adotado o termo **módulo** para representar estas 3 entidades. Para fazer a diferença entre a especificação e a implementação de uma API são utilizados os modificadores de acesso *private* ou *protected* para identificar as implementações e o modificador *public* para identificar as especificações. Então, pode-se afirmar que módulos *public* representam a API, visto que são estes que poderão ser utilizados pelos clientes. De fato, a maioria dos clientes de APIs deve conhecer um pouco do que está dentro de uma API, para serem capazes de utilizá-las [Tulach 2008]. Por isto, além dos módulos acima citados, outra peça fundamental de uma API é sua documentação. Existem diversas ferramentas capazes de gerar essa documentação de forma automática. Em Java, a mais usada é o Javadoc. Leslie [Leslie 2002] afirma que em muitos casos, o Javadoc é a única documentação disponível de um sistema e associa a grande utilização deste modelo pelo pequeno esforço para gerá-lo. O Javadoc de uma API é gerado através de uma ferramenta incluída no *Java Development Kit* da Sun Microsystems, onde a API foi desenvolvida.

No contexto de APIs, existem dois papéis importantes: os clientes e os desenvolvedores de uma API. Os clientes são os desenvolvedores de aplicações que utilizam os serviços de uma API, enquanto que os desenvolvedores são os programadores que escreveram a implementação da API.

2.1. Usabilidade de APIs

Uma API é considerada estável quando não são realizadas mudanças freqüentes na mesma ou se forem necessárias mudanças, estas não devem afetar os códigos dos clientes [Tulach 2008]. Desta forma, é possível manter a independência entre os códigos dos clientes e dos desenvolvedores. Estas mudanças podem ser correções de erros ou criações de novas funcionalidades. Algumas destas mudanças podem até afetar os clientes. Entretanto, elas se tornam problemáticas quando ocorrem com freqüência e/ou afetam o cliente de forma que seu código deva ser, em grande parte, refeito.

Quando diferentes partes do mesmo sistema são tratadas separadamente, chamamos isto de modularização [Parnas 1972]. Neste caso, é necessário definir como o sistema será dividido em módulos. De acordo com o princípio proposto por Parnas, recomenda-se que as

implementações sejam ocultas e apenas as especificações sejam expostas. Logo, podemos inferir que as implementações são as partes de uma API que podem sofrer alterações de forma que não afetem o cliente, já que ele não tem acesso a esta parte da API. Por outro lado, as especificações são as partes da API que os clientes têm acesso e usam. Portanto, não devem sofrer alterações freqüentes ou suas alterações não devem afetar seus clientes [Parnas 1972, Larman 2001].

2.2. Por que utilizar APIs?

APIs são amplamente utilizadas por tornarem um aplicação mais modular, além de garantirem desenvolvimento mais ágil de aplicações, já que elas fornecem inúmeros serviços, como acesso a banco de dados, visualizações de informações, etc. Outro exemplo de API são as IDEs, como Eclipse [Gamma e Beck 2003] e Netbeans [Boudreau 2002], que disponibilizam inúmeros serviços que podem ser utilizados através de *plugins*, que funcionam como clientes para estas IDEs. O cliente não precisará criar um novo banco de dados ou IDE para usar no desenvolvimento de sua aplicação, sabendo que já existe uma API que disponibiliza esta funcionalidade. Segundo Tulach (2008) "(...) reusando o máximo possível, não escrevendo completamente o sistema, os times podem se concentrar no diferencial mais importante: a lógica de seu sistema".

Atualmente, aplicações usam diversas APIs *open source* disponíveis [Tulach 2008], que vão de bibliotecas básicas de *C* até servidores e navegadores web. Em Java, temos exemplos de bibliotecas e conjunto de ferramentas *open source* como JUNG [Madahain et al. 2007], Prefuse [Heer et al. 2005], JFreeChart [Lee et al. 2007], etc. Estas bibliotecas e ferramentas possuem APIs que são usadas por aplicações clientes, ou seja aplicações que usam os serviços providos por elas. Por isso é necessário que se faça uma avaliação durante a criação de uma API, pois clientes podem achar que uma determinada API não é tão simples de usar e todo o desempenho realizado durante o desenvolvimento daquela biblioteca será em vão.

Devido ao grande número de APIs disponíveis para a realização de um mesmo serviço, é importante permitir ao usuário identificar facilmente qual dentre as opções é menos complexa, logo mais fácil de usar. É justamente isto que a ferramenta MetriX faz, conforme será descrito na próxima seção.

3. Complexidade e Métricas de Software

3.1. Complexidade de Software

Provavelmente a mais antiga e mais obvia noção de complexidade é a quantidade de sentenças em um programa [Weyuker 1988]. A principal vantagem desta medida é sua simplicidade. Embora haja diferentes formas de se definir sentença de um programa, uma vez escolhida a forma, a tarefa de computar a complexidade do programa se torna direta e facilmente automatizada. Esta definição inicial sugere que algo complexo, tem um grande número de propriedades (Bunge citado por Bandi [Bandi et al. 2003]). Abbott [Abbott 1994] estendeu a definição de Bunge e definiu a complexidade como sendo uma função das interações dos conjuntos de propriedades. Neste caso, o conjunto de propriedades é dado por objetos e classes, os métodos e atributos, sendo assim, a complexidade de uma classe é uma função da interação entre os métodos e os atributos.

Como descrito anteriormente, uma API é o conjunto de especificações de interfaces, métodos e classes. Logo, a complexidade de uma API é a soma da complexidade de cada um destes módulos. Neste trabalho, a complexidade é calculada a partir de métricas de software.

3.2. As Métricas de Complexidade de Bandi

Métricas são valores numéricos de representação de dados de software e devem ser significativas, ou seja, deve haver uma razão para que as métricas sejam coletadas [Archer 1998]. Algumas métricas de software são calculadas a partir do código fonte dos sistemas, como por exemplo, Linhas de Código (LOC). Entretanto, no caso de APIs, nem sempre se tem disponível o código fonte, o que torna impossível a utilização de métricas tradicionais. Além disso, por definição, uma API consiste apenas na especificação da interface de um conjunto de serviços que pode ser oferecido por um sistema de software. Por isso, as métricas utilizadas pela ferramenta são métricas independentes de código fonte. Mais precisamente, este trabalho utiliza as métricas propostas por Bandi [Bandi et al. 2003] e por Boxall e Araban [Boxall e Araban 2004].

As métricas propostas por Boxall e Araban automatizadas pela MetriX são as seguintes: (i) Número de Métodos (NM) que como o próprio nome diz, calcula a quantidade de métodos contidos na API; (ii) Número de Argumentos (NA) que representa a quantidade de argumentos encontrados nos métodos da API; e finalmente (iii) Argumentos por Método (APM) que indica a média de argumentos por método encontrados no sistema.

As métricas propostas por Bandi são Nível de Interação , Tamanho da Interface e Complexidade do Argumento da Operação. Estas métricas focam na interface de um módulo. A base fundamental para estas métricas parte do pressuposto que quanto maior a interface, maior o escopo para interações (diretas) e estas interações aumentam a complexidade da API. Este pressuposto é consistente com as noções de complexidade discutidas acima e são baseadas no tamanho (peso) de parâmetros e atributos, apresentados na Tabela 1.

Cada uma das métricas é explicada a seguir. Para o cálculo dos pesos nas métricas a seguir, será utilizado a simbologia $P(p_n)$ e $P(a_m)$ para representar o Peso (P) do **n**-ésimo parâmetro (p) e o Peso (P) do **m**-ésimo atributo (a), respectivamente. Logo se um parâmetro (p) ou atributo (a) for do tipo *Integer*, P será 1.

Tabela 1: Tipos e Pesos dos atributos/parâmetros

Tipo	Peso
Boolean	0
Character	1
Integer	1
Byte	0
Short	2
Long	2
Float	2
Array	3

A métrica denominada **Nível de Interação** (IL) indica a quantidade de potenciais interações (diretas) que podem ocorrer em um sistema, classe ou método, ou seja, esta métrica calcula a quantidade de interações que podem ocorrer entre parâmetros de um método e atributos de sua classe. Segundo Bandi, espera-se que quanto maior o Nível de Interação, maior será a dificuldade em determinar a forma de implementar ou modificar um projeto. Ou seja, esta métrica tem importância para o **desenvolvedor** que implementa ou o **mantenedor** que mantém a implementação de uma API. Isso se dá pelo fato de que, se mudarmos um dos atributos de uma classe e este atributo for usado em diversos métodos, todos estes métodos poderão sofrer alterações.

A partir destes valores, para Bandi o IL é dado por:

$$IL = NA \cdot NP + \sum_{n=1...NA} \sum_{m=1...NP} (P(p_n) \cdot P(a_m))$$

Tamanho da Interface (IS) é a métrica que demonstra a média de informação que é transmitida dentro e fora do encapsulamento de uma classe. Segundo Bandi, espera-se que quanto maior o Tamanho da Interface maior será a dificuldade de compreender como selecionar e usar corretamente os serviços fornecidos por uma classe. Ou seja, esta métrica fornecerá uma complexidade relevante para *clientes* de APIs, que como descrito anteriormente, usam os serviços providos pela API. O cálculo do IS de um método é dado por:

$$IS = NP + \sum_{n=0...NP} P(p_n)$$

A métrica **Complexidade do Argumento da Operação** (OAC) é a métrica mais simples proposta por Bandi. Esta métrica leva em consideração apenas o peso (P) de cada parâmetro (p), e é dada pela soma destes pesos.

$$OAC = \sum_{n=0...NP} P(p_n)$$

As métricas obtidas através da ferramenta representam a complexidade de métodos, classes e assim por diante. Seus significados vão depender de quem os observa, por exemplo, um cliente poderá identificar qual API é mais simples de usar e qual é mais complexa.

Na MetriX, as métricas acima são utilizadas para o cálculo da complexidade das APIs e, partir da complexidade são geradas as visualizações da ferramenta, que serão detalhadas na seção seguinte.

3.3. Exemplo de Cálculo das Métricas de Bandi

Tomando como base uma classe hipotética C (representada na Figura 1) que possui as seguintes características:

- 8 atributos do tipo *float* (x_1, y_1, x_2, y_2, x_3, y_3, x_4 e y_4)
- 1 método (*m*)
- o método *m*, por sua vez, possui 3 parâmetros[2]. 2 do tipo *float* e 1 do tipo *boolean*

[2] Para Bandi o valor devolvido por um método, se existir, também é considerado um parâmetro.

(x, y e boolean)

A Figura 1 exibe as interações que ocorrem dentro da classe C, ou seja, as relações que podem ocorrer entre os parâmetros do método m e os atributos da classe C.

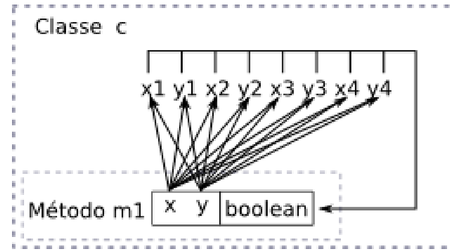

Figura 1: Interação entre parâmetros do método m e atributos da classe C

O **Nível de Interação** (IL) de uma classe é dado pela soma do IL dos métodos que compõem esta classe. Portanto, como a classe C possui apenas o método m, o IL da classe C (IL_C) é igual ao IL do método m (IL_m). Por sua vez, o IL_m é obtido da seguinte forma:

$$\begin{aligned}
IL_m &= NA \cdot NP + \sum_{n=1...NA} \sum_{m=1...NP} (P(p_n) \cdot P(a_m)) \\
&= 8 \cdot 3 + P(p_1) \cdot P(a_1) + \cdots + P(p_1) \cdot P(a_8) + \cdots + P(p_3) \cdot P(a_8) \\
&= 24 + P(x) \cdot P(x_1) + \cdots + P(x) \cdot P(y_4) + \cdots + P(boolean) \cdot P(y_4) \\
&= 24 + 2 \cdot 2 + \cdots + 2 \cdot 2 + \cdots + 0 \cdot 2 \\
&= 24 + 8 \cdot (2 \cdot 2) + 8 \cdot (2 \cdot 2) + 8 \cdot (0 \cdot 2) \\
&= 24 + 32 + 32 + 0 \\
&= 24 + 64 \\
IL_m &= 88
\end{aligned}$$

Como $IL_C = IL_m$, $IL_C = 88$.

De maneira análoga, o **Tamanho da Interface** (IS) de uma classe é dado pela soma do **IS** dos métodos que compõem esta classe. Portanto, o $IS_C = IS_m$. E IS_m é dado por:

$$\begin{aligned}
IS_m &= NP + \sum_{n=1...NP} P(p_n) \\
&= 3 + P(p_1) + P(p_2) + P(p_3) \\
&= 3 + P(x) + P(y) + P(retorno) \\
&= 3 + 2 + 2 + 0 \\
IS_m &= 7
\end{aligned}$$

Como $IS_C = IS_m$, $IS_C = 7$.

A **Complexidade do Argumento da Operação** (OAC) da classe C (OAC_C) é a soma do OAC dos seus métodos, logo, $OAC_C = OAC_m$. E OAC_m é dado por:

$$\begin{aligned}
OAC_m &= \sum_{n=0..NP} P(p_n) \\
&= 2 + 2 + 0 \\
OAC_m &= 4
\end{aligned}$$

Logo, $OAC_C = 4$.

4. A Ferramenta Metrix

A ferramenta proposta neste trabalho tem como objetivo fornecer ao usuário uma visualização que torne a tarefa de análise da complexidade de uma API mais simples. Através desta análise, é possível decidir qual API utilizar em um projeto, ou ainda avaliar possíveis problemas de manutenção futuros devido à alta complexidade de uma dada API. A ferramenta também pode auxiliar um gerente de projeto a alocar pessoas para uma tarefa de correção de um erro baseado na complexidade da(s) classe(s) onde a tarefa será realizada.

4.1. Intervalos de Complexidade

Segundo Lanza [Lanza 2005], qualquer que seja a métrica utilizada, deve-se saber o que é muito alto, muito baixo, alto ou baixo. Isto é necessário para que possam ser realizadas comparações entre as métricas encontradas. Além disso, um ponto crucial para se trabalhar com métricas é ser capaz de interpretar os valores corretamente. Lanza também afirma que não há como definir limiares perfeitos como referência. Entretanto, pode-se definir limiares explicáveis, ou seja, valores que podem ser escolhidos baseados em argumentos razoáveis. Como os limiares são fundamentais para análises baseadas em métricas, durante o desenvolvimento da MetriX, foi feita uma análise de 12 APIs diferentes. A idéia era criar um banco de dados com estatísticas sobre a complexidade de diversas APIs. Desta forma, quando o usuário analisa uma API, indiretamente e automaticamente ele está avaliando a usabilidade desta API em comparação com outras. Isto é, o banco de dados permite a efetiva interpretação das métricas calculados. Em resumo, os limiares utilizados pela MetriX representam as métricas encontradas nas APIs analisadas.

As doze APIs foram escolhidas de forma que não fossem todas do mesmo propósito. Assim foram analisadas no total 8155 classes, cujas complexidades variaram de 0 a $1,69 \times 10^{24}$, com um total de 2380 valores diferentes. A partir destes dados foi construído um histograma: o eixo X representava a soma das complexidades propostas por Bandi para um determinada classe e o eixo Y indicava o número de classes com uma determinada complexidade. Analisando estes dados utilizando o programa Arena[3] foi possível identificar que eles seguiam uma distribuição weibull. Além disso, a partir do histograma foi possível observar que a maioria das classes tinham baixa complexidade. Ainda com base no histograma, foi calculada a freqüência de ocorrência dos valores para cada métrica encontrada. Assim, pôde-se organizar as métricas em ordem crescente e após isto, estas métricas foram divididas em 10 intervalos (Tabela 2). Estes 10 intervalos foram denominados *dectril*[4].

Tabela 2: Dectril utilizado pela MetriX.

0	244	600	1402	5268	36268	1688094	2147484244	4498507340	13035189028

Levando em consideração estes intervalos, foram definidas cores que os representariam nas visualizações (Figura 2).

Figura 2: Intervalo de cores que representa o Dectril nas visualizações da Ferramenta

[3] www.arenasimulation.com

[4] Uma analogia à técnica *quatril*, que divide uma amostra em 4 intervalos diferentes.

4.2. Visualizações Utilizadas na Ferramenta

A ferramenta gera três visualizações diferentes, sendo que duas delas (**TreeMap** e **StarPlot**) são baseadas na complexidade, e a terceira (**Grafo de Dependência**) que exibe as relações entre classes e entre classes e métodos. A visualização **TreeMap** é capaz de exibir duas métricas, que são 1) a soma das métricas **IL**, **IS** e **OAC**, denominada **TOTAL** e 2) **NM**, estas métricas são exibidas da seguinte forma: o tamanho da classe na visualização representa o **NM** e a cor representa a complexidade baseada no **TOTAL**. A visualização **StarPlot** pode exibir as seis métricas, sendo cada métrica representada por uma ponta da estrela.

A visualização **TreeMap** (Figura 3) é um método de visualização para preenchimento de espaço capaz de representar grandes quantidades hierárquicas de dados quantitativos em uma exibição compacta [Bederson 2002]. Esta visualização pode ser executada em 2 níveis: API e pacote. Esta funcionalidade é interessante para que o usuário possa observar com mais detalhes caso tenha interesse em uma pacote específico. Nesta visualização são exibidas todas as classes contidas no contexto (API ou Pacote). Na visualização TreeMap as **linhas pretas** delimitam os **pacotes** e as **linhas brancas** as **classes**. Os retângulos que representam as classes possuem cores, que representam o nível de complexidade delas, seguindo as escalas (de cor e de complexidade) apresentadas na seção 4.1. O tamanho destes retângulos representa a quantidade de classes que um pacote possui ou a quantidade de métodos que a classe possui. Através do tamanho do retângulo também é possível perceber a concentração de classes e métodos. Desta forma, se um pacote ocupa 1/5 da visualização, isso quer dizer que aquele pacote possui 1/5 das classes daquela API.

Na Figura 3 é exibida a avaliação da API do OpenCOM [Coulson et al. 2008], que é um framework utilizado para desenvolvimento de sistemas baseados em componentes, conseqüentemente em aplicações ubíquas.

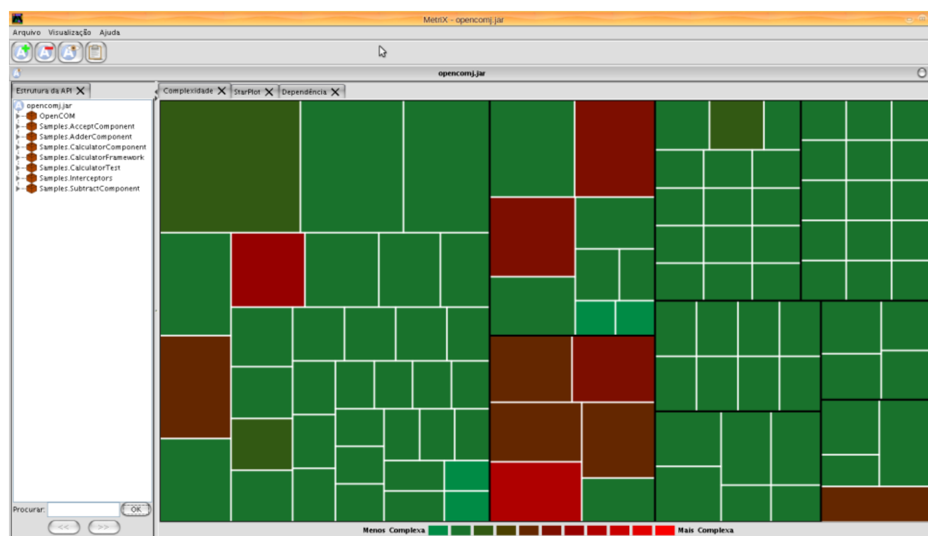

Figura 3: Visualização TreeMap gerado pela MetriX avaliando o OpenCOM

A visualização **StarPlot** (Figura 4) utilizada na MetriX foi criada para a ferramenta Sargas [Júnior 2008]. Esta visualização pode apresentar os 3 níveis, API, Pacote e Classe. Em nível de API, são exibidas estrelas que representam os pacotes da API (Figura 3). Quando gerado o StarPlot de um pacote, são exibidas estrelas que representam as classes daquele pacote.

Em nível de classe são exibidos os métodos desta classe. Na visualização StarPlot é possível notar um círculo concêntrico à estrela. Este círculo representa a média entre as métricas daquela visualização [Júnior 2008], ou seja, será gerada a visualização de uma API, o círculo é a média das métricas dos pacotes exibidos na visualização da API. Logo se um dos lados da estrela ultrapassar este círculo pode-se dizer que aquela métrica está acima da média.

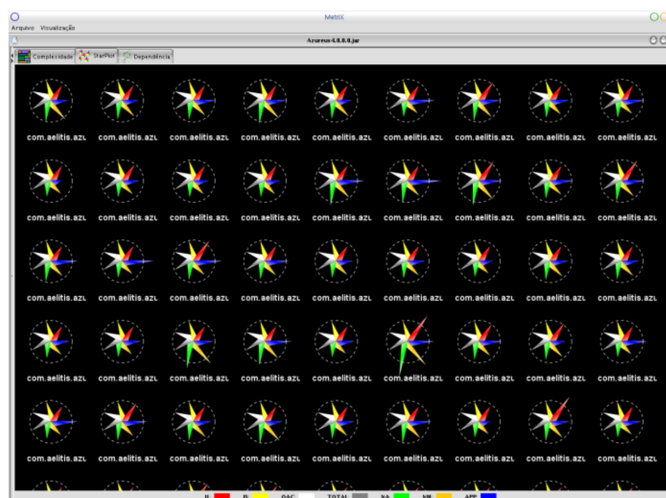

Figura 4: Visualização StarPlot gerada pela MetriX

O **Grafo de Dependência** (Figura 5) exibe a relação que existe entre classes e entre classes e métodos. Nesta visualização as classes são representadas por círculos e os métodos pela letra 'M'. Em nível de API e pacote são exibidas as dependências entre classes. Quando o grafo é gerado em nível de classe é exibida a relação entre métodos desta classe, com as demais classes da API. Nesta visualização, as classes aparecem com cores diferentes e estas cores representam os pacotes destas classes, ou seja, cada pacote será apresentado de uma cor. A legenda desta visualização exibe a relação de cores com o nome dos pacotes.

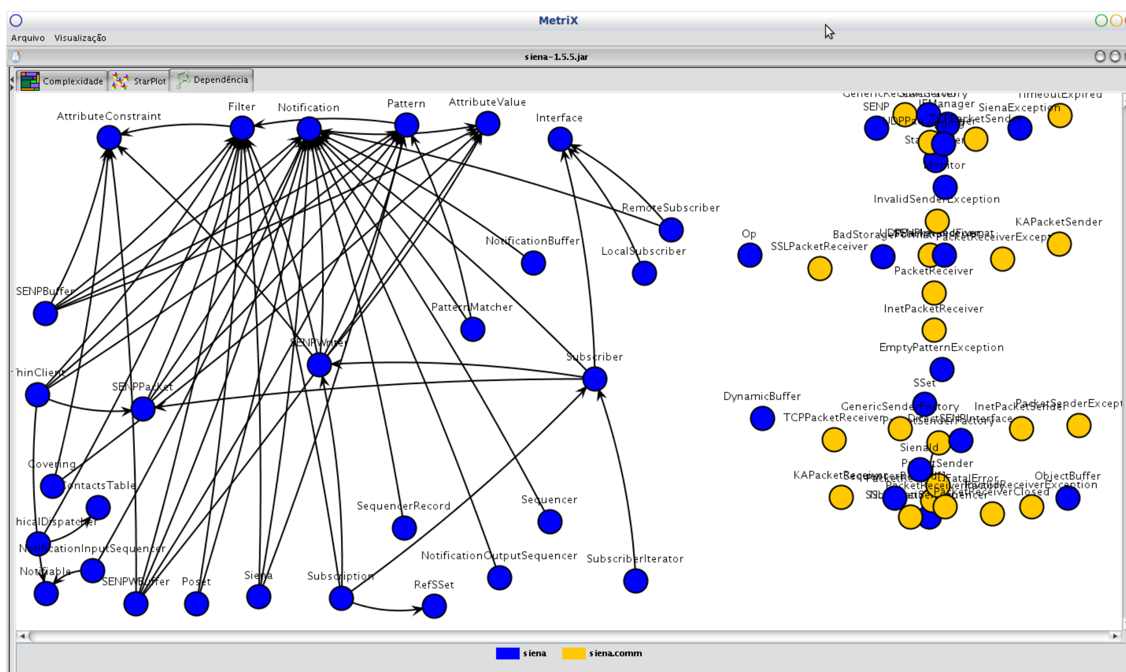

Figura 5: Visualização do Grafo de Dependência

As arestas do grafo representam as relações (de dependência). O grafo de dependência é um grafo direcionado, ou seja, as arestas são setas. Na origem das setas estão nós que representam classes ou métodos que dependem das classes que estão no destino da seta. No destino das setas serão encontradas apenas classes, isto se dá pelo fato de que em hipótese alguma uma classe dependerá de um método. Para notar a importância de uma classe no projeto, pode-se tomar como base o grau de entrada[5] do vértice, ou seja, quanto maior o número de setas que incidem em um nó, mais importante esta classe será, pois muitas classes ou métodos dependem dela.

Esta visualização pode ser utilizada para analisar o impacto da mudança em uma classe da API. Por exemplo, se um método tem alguma relação com uma classe (como parâmetro ou tipo de retorno), uma mudança nessa classe pode afetar o método, pois este pode usar algum método ou atributo da classe. Outra utilidade desta visualização é para uma pessoa que precise utilizar um método de uma classe. Através desta visualização, ela poderá saber quais classes precisa ou deveria entender para utilizar este método da melhor forma possível.

A ferramenta permite que o usuário tenha a visualização de duas ou mais APIs simultaneamente a fim de que o usuário possa compará-las visualmente a partir da ferramenta. No caso da Figura 5 compara as APIs Jung (do lado esquerdo) e Prefuse (do lado direito). A partir de uma análise visual inicial pode-se perceber que a API Prefuse é mais complexa que a API Jung, pois possui mais classes "Vermelhas".

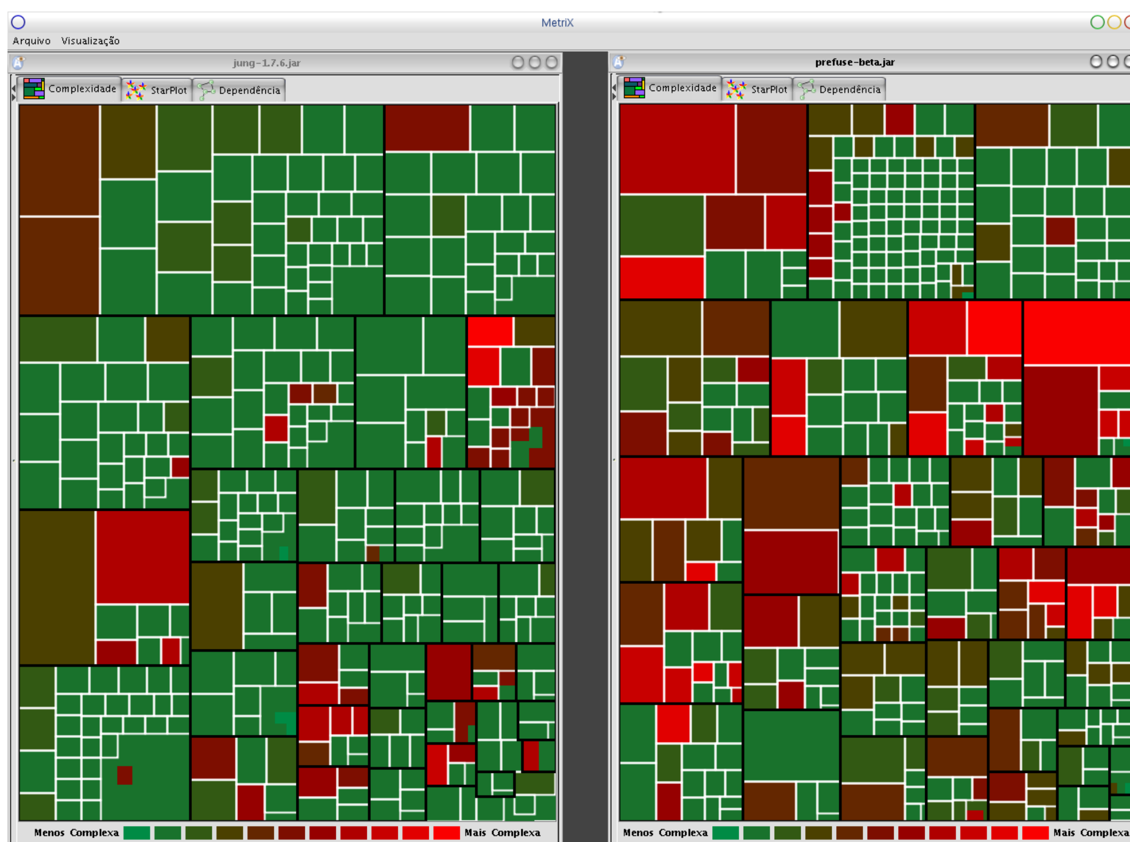

Figura 6: Comparação entre as visualizações das APIs do Jung e do Prefuse

[5] Quantidade de arestas que incidem em um nó.

5. Limitações da Abordagem

A ferramenta Metrix, apesar de possuir uma estrutura capaz de analisar qualquer API OO, na versão atual, permite analisar apenas APIs desenvolvidas em Java. Entretanto, para que sejam analisadas APIs em novas linguagens de programação OO basta criar um *parser* que realize esta tarefa, ou seja, este parser ficará encarregado de analisar a API e adaptá-la ao modelo de dados utilizado pela MetriX.

Além disso, apesar da MetriX analisar APIs OO, ela utiliza as métricas propostas em [Bandi et al. 2003], que não estão totalmente adaptadas a OO, logo o cálculo da complexidade de APIs realizado pela MetriX não leva em consideração técnicas como Herança e Polimorfismo. Como estas técnicas são utilizadas em linguagens OO, conseqüentemente também são utilizadas em APIs desenvolvidas utilizando estas linguagens.

Finalmente, a MetriX ainda não tem um desempenho satisfatório quando são avaliadas APIs de grande porte, como é o caso da API do cliente torrent Azureus, demorando cerca de 1 minuto para analisar a API e persisti-la no Banco de Dados. Após isso, ela demora aproximadamente mais 30 segundos para gerar a visualização da mesma API.

6. Trabalhos Relacionados

Recentemente, Stylos [Stylos 2006, 2008] e Ellis [Ellis 2007] têm realizado pesquisas para avaliar a usabilidade de APIs utilizando uma abordagem diferente baseada em avaliação de usabilidade [Nielsen et al. 1990] de APIs. Por exemplo, em 2006, Stylos avaliou métodos construtores com parâmetros (opcionais) ou sem parâmetros. Uma limitação desta abordagem é que não levava em consideração os demais métodos da classe, que também são utilizados por clientes de APIs, logo influenciam em sua complexidade e usabilidade. Em outro trabalho, Stylos [Stylos et al. 2007] realizou um estudo sobre a relação entre o uso do padrão de projeto *Factory* e uso de métodos construtores no projeto de API, e como o uso de cada um implica na usabilidade da API. Os resultados deste estudo sugerem que o uso do padrão *Factory* [Gamma 1995] prejudica a usabilidade da API, pois o cliente de uma API gasta mais tempo para entender e usar uma API do que se usasse um método construtor. Finalmente, em um trabalho de 2008, Stylos e Myers avaliaram como o posicionamento de um método implica no aprendizado de uma API. O objetivo deste trabalho era o melhor aumento do potencial de APIs, entendendo o que as faz difíceis para programadores e como resolver estes problemas (corrigindo as APIs, a documentação ou com novas ferramentas de programação).

Apesar da similaridade destes trabalhos, o trabalho aqui proposto apresenta como diferencial o fato de utilizar uma abordagem tradicional da comunidade de engenharia de software, métricas de software [Fenton e Pfleeger 1997].

7. Considerações Finais

Este trabalho teve como objetivo descrever a ferramenta MetriX, que automatiza a tarefa de cálculo de complexidade de APIs OO, utilizando métricas específicas para APIs, visto que APIs não têm seu código disponível. A partir destas métricas é possível gerar visualizações com dados relevantes para desenvolvedores e clientes de APIs. Estas visualizações podem auxiliar em tomadas de decisões, como alocação de recursos e definição de custo para tarefa de manutenção e, principalmente, na escolha de uma API a ser utilizada para uma determinada

atividade. A análise da complexidade de APIs é um fator importante para permitir a construção de sistemas fidedignos de alta complexidade, sejam eles ubíquos ou não.

A ferramenta aqui proposta mostrou-se válida em testes iniciais realizados por alunos da Universidade Federal do Pará (UFPA), que já haviam desenvolvido uma API de uma ferramenta de processo de software. A MetriX comprovou que as classes que eles consideravam complexas foram realmente apresentadas pela ferramenta como complexas.

Atualmente, a ferramenta é utilizada por um doutorando da Universidade da Califórnia, Irvine (UCI), para auxiliar na tarefa do cálculo da complexidade, bem como a análise de dependência entre as classes de APIs desenvolvidas em Java. Neste caso MetriX tem sido utilizada para comparar APIs de servidores de notificação de eventos. Durante a utilização pelo doutorando, estão sendo obtidos relatos positivos e negativos sobre a ferramenta, que têm servido para que a ferramenta seja sempre aprimorada e adaptada as necessidades dos usuários. Pretende-se dar continuidade ao desenvolvimento da ferramenta, principalmente adaptando as métricas propostas por Bandi aos conceitos de OO e criando novas métricas que possam ser aplicadas a APIs e automatizadas através da MetriX, bem como definição dos pesos de outros tipos de parâmetros/atributos existentes em linguagens OO. A partir destas novas métricas e novos pesos, novas APIs serão analisadas e a partir destas análises, novos limiares serão definidos. Também pretende-se aumentar o número de visualizações, para que sejam exibidas as informações mais relevantes possíveis. Esta etapa inclui encontrar, não quaisquer visualizações, mas visualizações que sejam capazes de destacar detalhes que podem não ser percebidos pelas visualizações utilizadas atualmente.

Outro objetivo para trabalhos futuros é melhorar o *parser* utilizado para a linguagem Java, para que seja capaz de identificar também as exceções, já que estas têm papel importante, principalmente na geração do grafo de dependência. Além do melhoramento do *parser* atual, pretende-se aumentar o número de *parsers*, para superar a limitação da ferramenta, que atualmente reconhece apenas APIs implementadas na linguagem Java. O objetivo é fazer o maior número possível de *parsers*, para que a ferramenta possa ser utilizada em APIs que utilizam as mais variadas linguagens OO.

Pretende-se também, melhorar o desempenho da ferramenta. Apesar de satisfatório, o desempenho ainda pode ser melhorado, através de algumas técnicas de programação (processamento paralelo e otimização no código do *parser*) que começaram a ser estudadas, porém ainda não foram colocadas em prática.

Como continuação deste projeto, pretende-se realizar mineração de dados em documentações de APIs existentes e em exemplos de utilização de APIs encontrados através de máquinas de busca. Esta tarefa será realizada com o objetivo de encontrar padrões de utilização de uma API e através destes padrões será possível identificar se as classes mais complexas são utilizadas ou não por clientes. Por exemplo, se for encontrado que clientes utilizam apenas classes ”verdes” a complexidade da API não é tão grande, mesmo ela possuindo diversas classes ”vermelhas”.

Agradecimentos

Este trabalho tem apoio financeiro do CNPq através do Edital CT-Info 550931/2007-4.

Referências

Abbott, D. (1994) "A Design Complexity Metric for Object-Oriented Development", Dissertação (Mestrado) — Clemson University.

Archer, C. B., Stinson, M. C. (1998). "Object-oriented software product metrics" (tutorial). In SIGCPR '98: Proceedings of the 1998 ACM SIGCPR conference on Computer personnel research, pages 305–306, New York, NY, USA. ACM.

Bandi, R. K., Vaishnavi, V. K., Turk, D. E. (2003). "Predicting maintenance performance using object-oriented design complexity metrics". IEEE Trans. Softw. Eng., IEEE Press, Piscataway, NJ, USA, v. 29, n. 1, p. 77–87, 2003. ISSN 0098-5589.

Bederson, B. B., Shneiderman, B., Wattenberg, M. (2002) "Ordered and quantum treemaps: Making efective use of 2d space to display hierarchies", ACM Trans. Graph., ACM, New York, NY, USA, v. 21, n. 4, p. 833 – 854, ISSN 0730-0301.

Boudreau, T., Glick, J., Spurlin, V. (2002) Netbeans: the Definitive Guide. O'Reilly & Associates, Inc.

Boxall, M. A. S., Araban, S. (2004) "Interface metrics for reusability analysis of components". Australian Software Engineering Conference, v. 20, n. 6, p. 40–51.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). "A generic component model for building systems software". ACM Trans. Comput. Syst. 26, v. 1, p. 1–42.

Ellis, B., Stylos, J., Myers, B. (2007) "The factory pattern in API design: A usability evaluation", IEEE Computer Society, Washington, DC, USA, p. 302 – 312.

Fenton, N.E., Pfleeger, S.L. (1997) "*Software Metrics: A Rigorous & Practical Approach*". Second ed., London: PWS Publishing Company.

Gamma, E., Beck, K. (2003) "Contributing to Eclipse: Principles, Patterns, and Plugins". Addison Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R.,Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Heer, J., Card, S. K., Landay, J. A. (2005) "Prefuse: a toolkit for interactive information visualization". SIGCHI Conference on Human Factors in Computing Systems. CHI '05. ACM, p. 421–430

Júnior, S. F. de S., Souza, C. R. B. de. (2008) "Visualização integrada de múltiplas métricas de redes sociais", Workshop on Information Visualization and Analysis in Social Networks, WIVA, Campinas, São Paulo, Brasil. ISSN 9788576692058.

Lanza, M., Marinescu, R., Ducasse, S. (2005) "Object-Oriented Metrics in Practice". Secaucus, NJ, USA: Springer-Verlag New York, Inc.. ISBN 3540244298.

Larman, C. (2001) "Protected variation: The importance of being closed", IEEE Software, IEEE Computer Society, Los Alamitos, CA, USA, v. 18, n. 3, p. 89 – 91. ISSN 0740-7459.

Lee, Y., Yang, J., Chang, K. H. (2007) "Metrics and Evolution in Open Source Software".

Seventh international Conference on Quality Software IEEE Computer Society, Washington, DC, 191–197.

Leslie, D. M. (2002) "Using javadoc and XML to produce api reference documentation", ACM, New York, NY, USA, p. 104 – 109.

Madahain, J. O., Fisher, D., Smyth, P., White, S., Boey, Y., (2007) "Analysis and Visualization of Network Data using JUNG". Journal of Statistical Software, n.2

Nielsen, J., Molich, R. (1990) "Heuristic evaluation of user interfaces", CHI '90: SIGCHI conference on Human factors in computing systems, ACM, Seattle, Washington, United States, p. 249 – 256. ISBN 0-201-50932-6

Parnas, D. L. (1972) "On the criteria to be used in decomposing systems into modules. Commun", ACM, ACM, New York, NY, USA, v. 15, n. 12, p. 1053 – 1058. ISSN 0001-0782.

Simon, H. A. (1996). The Architecture *of Complexity: Hierarchical Systems. The Sciences* of the Artificial. Cambridge, MA, The MIT Press: 183-216.

Stylos, J. (2006) "Informing API design through usability studies of API design choices: A research abstract", IEEE Computer Society, Washington, DC, USA, p. 246 – 247.

Stylos, J., Myers, B. A. (2008) "The implications of method placement on API learnability". ACM, New York, NY, USA, p. 105 – 112.

Tulach, J. (2008) "Practical API Design: Confessions of a Java Framework Architect", Berkeley, CA, USA: Apress. ISBN 1430209739, 9781430209737.

Weiser, M. (1991). "The computer for the 21st century". Scientific American, v. 265, n. 3, p. 94–104.

Weyuker, E. J. (1988) "Evaluating software complexity measures". IEEE Trans. Softw. Eng., IEEE Press, Piscataway, NJ, USA, v. 14, n. 9, p. 1357–1365, ISSN 0098- 5589.