Evaluating the Implications of a Package Design Principle upon Software Maintainability

Article · September 2010

DOI: 10.1109/SBES.2010.24

2 authors, including:

Márcio de Oliveira Barros

Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

94 PUBLICATIONS **545** CITATIONS

SEE PROFILE

Avaliando as Implicações do Princípio de Projeto Common-Closure sobre a Manutenção do Software

Marcelo de F. Costa
PPGI/UNIRIO
Av. Pasteur, 458
Rio de Janeiro, RJ Brasil
marcelo.costa@uniriotec.br

Márcio de O. Barros
PPGI / UNIRIO
Av. Pasteur, 458
Rio de Janeiro, RJ Brasil
marcio.barros@uniriotec.br

Abstract – Software systems are evolvable constructs which must be constantly changed to remain useful. However, the effort required to support this evolution is usually huge, growing as the system ages and is changed in less-than-controlled ways. Software design principles propose ways to organize the basic components of these systems in order to accommodate change and reduce the overall maintenance effort. In this paper, we address the Common-Closure package design principle, proposing a technique to organize the classes comprising a system into packages according to this principle. We present the results of an experimental evaluation to ascertain whether the adoption of the Common-Closure principle improves a set of software design metrics.

Resumo – Sistemas de software devem ser constantemente modificados para que permaneçam úteis. Entretanto, o esforço necessário para suportar esta evolução é grande, aumentando na medida em que o sistema envelhece e é modificado de maneira pouco controlada. Princípios de projeto de software propõem maneiras de se organizar os componentes básicos desses sistemas, a fim de acomodar mudanças e reduzir o esforço de manutenção. Neste artigo, propomos uma técnica para organizar as classes que compõem um sistema em pacotes de acordo com o princípio de projeto de pacotes Common-Closure. Apresentamos o resultado de um estudo experimental para verificar se a adoção deste princípio melhora um conjunto de métricas de projeto de software.

Palavras-chave: *princípios de projeto de software, manutenção de software, coesão de pacotes, sistemas de informação*

I. INTRODUÇÃO

Sistemas de software têm de evoluir ou correm o risco de perder sua capacidade de apoiar os processos de negócio para os quais oferecem suporte, uma vez que estes processos se encontram em constante mudança [1]. No entanto, a manutenção de sistemas de larga escala é uma tarefa difícil, que consome muitos recursos humanos e financeiros. Atividades relacionadas com a adição de novas funcionalidades, adaptação do sistema para suportar novos dispositivos e plataformas, otimização e correção de erros tornam-se mais difíceis com a passagem do tempo, conforme o sistema cresce de forma pouco controlada [8].

Muitos autores [2] [5] [6] [8] [10] propõem princípios de projeto, que são diretrizes que orientam os desenvolvedores durante a fase de projeto do ciclo de vida do software, a fim de atenuar os efeitos do crescimento descontrolado do sistema que está sendo desenvolvido. Estes princípios estabelecem uma série de estratégias para organizar os elementos que compõem o software, de tal maneira que mudanças antecipadas possam ser introduzidas com esforço aceitável em uma fase mais avançada do seu ciclo de vida.

Sistemas orientados a objetos são compostos de classes, que são agrupadas em pacotes para permitir a decomposição sistemática da estrutura conceitual subjacente ao projeto. De forma análoga, os princípios de projeto orientados a objetos são divididos em duas categorias: princípios baseados em classes e princípios baseados em pacotes. Enquanto os princípios de projeto baseados em classes propõem formas de organizar as classes que compõem o sistema para resolver problemas estruturais recorrentes, princípios baseados em pacotes propõem estratégias para organizar um (grande) número de classes em pacotes de acordo com um conjunto de objetivos. Por exemplo, o princípio de projeto de pacotes *Common-Closure* [2] afirma que as classes em um pacote devem ser "fechadas" contra os mesmos tipos de modificações. Assim, uma alteração que afete uma classe em um pacote deve, em última instância, afetar todas as classes do pacote e nenhuma classe de outro pacote.

Embora muitas diretrizes tenham sido pesquisadas para apoiar o desenvolvimento, manutenção e evolução de sistemas [7] [9] [10] [11], pouco se sabe sobre a adoção destas orientações por desenvolvedores. De fato, há alguma evidência de que as boas práticas não são seguidas. Por exemplo, autores [2] [3] advertem os desenvolvedores quanto à criação de ciclos de dependência. No entanto, estudos descobriram que não só os desenvolvedores criam ciclos regularmente em seus softwares, mas muitas vezes esses ciclos são grandes estruturas, envolvendo muitas classes [3]. Se uma determinada prática é recomendada, mas não é encontrada em softwares no mundo real, isso pode indicar que tal prática não foi aceita pela academia e/ou indústria. Tal rejeição pode ser devido à falta de conhecimento, falta de divulgação ou a ausência de evidências sobre os benefícios gerados pela prática para os desenvolvedores de software.

Neste artigo, abordamos o uso do princípio de projeto *Common-Closure* (PCC) para distribuir as classes de um software orientado a objetos em pacotes. Estamos interessados em avaliar se a utilização do PCC melhora o projeto de um sistema orientado a objetos. Para realizar esta avaliação, propomos uma técnica que distribui as classes do sistema em pacotes, a fim de alinhar a estrutura do código fonte do sistema ao PCC. A estrutura proposta pela técnica pode então ser comparada com a estrutura escolhida pelos desenvolvedores para avaliar os benefícios e restrições trazidos pelo PCC. Propomos um procedimento de coleta de dados para determinar quais classes foram modificadas conjuntamente ao longo da implementação de um projeto de software e um modelo de otimização para agrupar estas classes em pacotes. Finalmente, apresentamos os resultados de uma análise experimental *in silico* [21] para descobrir se a organização de classes proposta pela técnica baseada no PCC melhora um conjunto de métricas de projeto.

Além desta introdução, este trabalho está organizado em mais seis seções. Em seguida, apresentamos um conjunto de princípios de projeto de pacotes e analisamos como eles afetam a evolução e a manutenção do software. Na seção III definimos uma representação para o perfil de modificação das classes que compõem um sistema e uma estratégia para identificar as classes que são modificadas em conjunto. Na Seção IV apresentamos uma técnica para distribuir classes em pacotes de acordo com o PCC. Uma avaliação experimental da técnica proposta é apresentada na Seção V. Limitações são abordadas na Seção VI, enquanto trabalhos futuros e conclusões são apresentados na última seção.

II. PRINCÍPIOS DE PROJETO DE PACOTES

Um dos desafios enfrentados por arquitetos de software quando projetam um sistema é escolher um bom agrupamento para as classes que compõem o sistema em estruturas maiores, que funcionem como contêineres. A distribuição das classes em tais estruturas facilita a identificação das classes responsáveis por implementar uma determinada funcionalidade fornecida pelo software, possibilita uma navegação mais fácil entre as partes do software [4] e uma melhor compreensão do programa [12].

Linguagens de programação (como Java, C # e C++) suportam uma organização de alto nível das classes que compõem um sistema através de estruturas que agrupam os módulos de código-fonte. Tais estruturas são implementadas de várias maneiras, dependendo da linguagem escolhida. Algumas linguagens adotam a abordagem de definir "espaços de nomes" (*namespaces*) para criar divisões hierárquicas de classes. Outras linguagens mapeiam estas divisões para o armazenamento físico dos módulos de código-fonte, geralmente exigindo que todas as classes do mesmo grupo pertençam ao mesmo diretório no sistema de arquivos. Em última análise, todas estas implementações têm a mesma finalidade: permitir aos desenvolvedores agrupar pequenos componentes do programa (classes) em entidades maiores e coesas, de acordo com algum critério.

Um pacote na *Unified Modeling Language* (UML) é usado "para agrupar os elementos componentes de um sistema e fornecer um *namespace* para estes elementos" [5]. A escolha de um esquema de agrupamento é fortemente influenciada pelos tipos de classe que compõem o sistema e pelos relacionamentos entre as classes. Relacionamentos entre classes determinam os relacionamento entre pacotes: se uma classe no pacote A depende de uma classe do pacote B, observamos um relacionamento de dependência entre os pacotes A → B. Por conseguinte, as relações entre os pacotes podem ser alteradas através do deslocamento de classes entre eles ou alterando o código-fonte de uma determinada classe de forma a retirar suas ligações com as classes de outros pacotes.

Bay [6] sugere que uma boa prática para a construção de software orientado a objeto é manter as entidades (classes e pacotes) com um tamanho reduzido. Melton e Tempero [3] também sugerem que os arquitetos de software devem estabelecer um limite para o tamanho dos pacotes (em termos de número de classes) e que a arquitetura deve ser controlada de forma a evitar pacotes excessivamente grandes. Esta sugestão é reforçada por [13], que apresenta limites inferiores e superiores para o número de classes por pacote, juntamente com um valor médio dessa relação. Fora estes trabalhos, a literatura não é clara no tocante ao melhor tamanho para um pacote [3] [5] [6], às vezes apresentando opiniões discordantes (ver [3] e [5], por exemplo).

Relacionamentos entre pacotes devem ser abordados da mesma maneira que os relacionamentos entre classes são tratados pelos princípios de projeto baseados em classes. O pacote deve ser tão independente de outros pacotes quanto possível e relacionamentos circulares devem ser evitados [3] [5]. A noção de dependência entre os pacotes é importante porque é desejável que um pacote seja reutilizado de um programa para outro carregando um conjunto mínimo de classes extras para trabalhar no novo ambiente. Assim, se um pacote depende apenas de si mesmo, ele pode ser reutilizado de forma isolada. Por outro lado, se um pacote depende de vários outros pacotes, todos estes devem ser levados para o novo programa juntos com o pacote que é efetivamente desejado.

Muitos princípios de projeto tratam da coesão de pacotes fazendo uma relação direta entre o objetivo do pacote e as mudanças que ele sofre ao longo do ciclo de vida de desenvolvimento. O Princípio da Responsabilidade Única (PRU) [2] afirma que um pacote deve ter um objetivo único e bem definido e que as classes que o compõem devem ser estritamente relacionadas com este objetivo. O princípio *Open-Closed* estabelece que as entidades de software (classes, funções e assim por diante) devem ser organizadas de modo que as mudanças sejam implementadas pela adição de novo código (extensão), em vez de modificar partes do

código que já funciona [7]. Assim, as arquiteturas de software devem ser projetadas de modo que uma mudança em uma única classe não resulte em uma cascata de mudanças em classes dependentes.

Finalmente, o PCC declara que classes em um pacote devem estar sujeitas aos mesmos tipos de modificação [2]. Assim, uma alteração que afete uma classe em um pacote deve, em última instância, afetar todas as classes do pacote e nenhuma classe que pertença a outros pacotes. O PCC está fortemente relacionado à implantação e reutilização dos pacotes, lidando com as diferentes "razões" que geram as solicitações de mudanças em pacotes diferentes, de modo que depois de implementar uma mudança apenas os poucos pacotes que foram modificados por conta dela devem ser testados e implantados nos clientes.

III. MAPEANDO MODIFICAÇÕES NAS CLASSES

Para avaliar se um sistema de software foi desenvolvido de acordo com o PCC, é preciso determinar se as classes pertencentes a cada um dos pacotes estão sujeitas aos mesmos tipos de mudança. Para isto, usamos dados coletados de um sistema de controle de versão a fim de identificar quando as classes mudam. Se duas ou mais classes são atualizadas em um sistema de controle de versão nos mesmos intervalos de tempo, consideramos que elas mudaram devido ao mesmo motivo. Assim, para verificar se um sistema está alinhado com o PCC precisamos identificar as classes que mudam conjuntamente com mais freqüência e determinar se elas pertencem ao mesmo pacote.

Em nossa análise, os termos “pacote” e “diretório” são considerados equivalentes. Os pacotes estão intrinsecamente relacionados com os diretórios, de modo que os primeiros são uma estrutura para organizar classes que implementam a lógica da aplicação, enquanto os segundos são uma estrutura de apoio à organização lógica do sistema de arquivos que mantém o código-fonte da aplicação. A equivalência entre pacotes e diretórios é importante porque a maioria dos sistemas de controle de versão está estritamente relacionada com o sistema de arquivos (armazenamento de arquivos e diretórios), enquanto que os princípios de projeto (como o PCC) se referem a classes e pacotes.

Algumas linguagens de programação orientadas a objeto impõem uma relação estreita entre diretórios e pacotes. Em Java, por exemplo, cada diretório deve ser um pacote e os arquivos armazenados neste diretório contêm as classes pertencentes ao respectivo pacote. Mesmo linguagens que não impõe explicitamente essa restrição (C #, por exemplo) consideram boa prática manter todas as classes pertencentes a um pacote no mesmo diretório, de modo a garantir a organização do código-fonte [14]. Assim, uma relação direta entre pacotes e diretórios (cada diretório será tratado como um pacote específico) parece razoável para a maioria dos sistemas orientados a objeto.

Além disso, as classes e os arquivos também são tratados como equivalentes: enquanto um arquivo pode conter mais de uma classe, parece razoável que elas pertençam ao mesmo pacote e que mudem juntas no mesmo intervalo de tempo. Assim, cada arquivo será considerado como tendo uma única classe. Se um arquivo tiver mais de uma classe, assumiremos que estas classes trabalham em conjunto para a mesma finalidade, que são alteradas em conjunto e que pertencem ao mesmo pacote. Se um ou mais destes pressupostos são demasiadamente restritivos para um determinado arquivo de código fonte, as classes deste arquivo devem ser divididas em dois ou mais arquivos antes de se aplicar a técnica proposta.

Seja C o conjunto de classes compondo o projeto, com $c \geq 1$ elementos. Cada classe é unicamente identificada por seu nome[1]. Assim, cada elemento $c_i \in C$ é representado como $c_i = [nome_i]$.

Seja P o conjunto de pacotes usados no projeto, com $p \geq 1$ elementos. Cada pacote é unicamente identificado por seu nome e contém um conjunto de classes. Assim, cada elemento $p_j \in P$ é representado como $p_j = [nome_j, cp_j]$, onde $cp_j \subseteq C$.

Seja R o conjunto de revisões de classes, com $r \geq 1$ elementos. Cada elemento $r_k \in R$ é descrito pelo conjunto de classes modificadas durante a revisão (cr_k), o identificador da revisão (id_k), o desenvolvedor que fez a modificação contida na revisão (a_k), e a data/hora quando a operação de *commit* que gerou esta revisão foi executada (d_k). Assim, $r_k = [cr_k, id_k, a_k, d_k]$.

Para capturar dados sobre as alterações sofridas pelas classes que compõem um sistema, consultamos o *log* com o histórico de modificações fornecido pela ferramenta de controle de versão utilizada no desenvolvimento do sistema. O *log* descreve uma série de revisões sofridas pelas classes que compõem o sistema. Cada revisão indica que arquivos (classes) foram modificados no contexto da revisão, o desenvolvedor que a gerou e a data/hora da sua criação/modificação. Descrevemos as classes de acordo com seu perfil de mudanças, criando, para cada uma das classes, a representação que chamamos de vetor característico (VC).

Considerando um período de tempo contido dentro do período de desenvolvimento (isto é, a partir da data em que a primeira classe que compõe o projeto foi adicionada ao sistema de controle de versão até a data atual), o VC de uma classe representa as mudanças sofridas pela classe durante este período. O período de interesse é representado a partir uma data de início ($d_{INÍCIO}$) até a data corrente. Ele é dividido em intervalos iguais, consecutivos e não

[1] Uma vez que temos a equivalência entre classes e arquivos, em linguagens de programação que não impõem a restrição de unicidade do nome da classe sugerimos fazer com que seu nome seja igual ao caminho completo do arquivo contendo a classe (único, por definição).

sobrepostos, cada um sendo representado por uma célula no VC. A Figura 1 apresenta um exemplo de VC.

O tamanho do intervalo (Δ) que divide o período de interesse (horas, dias, semanas, meses, dentre outros) é um parâmetro da técnica proposta. Quanto menor for o intervalo de tempo, maiores serão os vetores característicos para as classes sob análise. Por exemplo, classes compondo um projeto de dois anos podem ser representadas por um VC mensal de 24 células ou por um vetor semanal de 108 células.

	T_0	T_1	T_2	T_3	…	T_N
classe X	1	0	0	0	…	1

Figura 1. Exemplo de vetor característico de uma classe

Cada célula do vetor característico (T_i) representa um intervalo de tempo em que a classe pode ter sofrido alguma alteração. Se a classe foi modificada em um dado intervalo, a célula correspondente assume um valor igual a um (1). Caso contrário, a célula assume valor igual a zero (0). O vetor característico de uma classe pode ser criado a partir do *log* que contém o histórico de alterações registradas no sistema de controle de versões, anotando-se as datas das revisões nas quais a classe foi modificada. Os vetores característicos de todas as C classes que compõem um sistema devem ser construídos respeitando o mesmo período e intervalo de tempo.

Classes que não mudaram muitas vezes no período de interesse contêm pouca informação para serem comparadas com outras classes alteradas mais freqüentemente. Assim, ao comparar as classes que devem pertencer ao mesmo pacote, um arquiteto pode selecionar o conjunto de classes participantes especificando um número mínimo de revisões (r_{min}) que devem ter ocorrido durante o período. Depois de organizar as classes mais modificadas em pacotes, as classes restantes podem ser transferidas para os mesmos pacotes das classes com as quais elas mudaram com maior freqüência.

Depois de criar um vetor característico para cada classe, esses vetores podem ser justapostos para formar uma matriz característica. Cada linha da matriz representa um intervalo de tempo em que as alterações podem ter ocorrido, enquanto cada coluna representa uma classe. Cada célula da matriz indica se a classe a que sua coluna se refere foi modificada no intervalo de tempo a que sua linha se refere. A Figura 2 mostra um exemplo de uma matriz formada pelos vetores característicos das diferentes classes que compõem um sistema hipotético.

Intervalo	**classe A**	**classe B**	**classe C**	**…**	**classe N**
T_1	1	0	1	…	1
T_2	1	0	0	…	1
T_3	1	0	1	…	0
T_4	1	1	1	…	0
…	…	…	…	…	…
T_N	0	1	1	…	0

Figura 2 – Exemplo de uma matriz característica

A matriz característica contém todas as informações relevantes sobre as mudanças sofridas por cada classe e é usada como base para identificar as classes que mudam em conjunto e, sendo assim, devem pertencer ao mesmo pacote de acordo com o PCC. Considerando que os vetores característicos são construções geométricas, definimos o Espaço das Classes como uma metáfora para ilustrar a espaço geométrico onde esses vetores são representados e onde a distância entre as classes pode ser calculada.

O Espaço das Classes possui uma dimensão para cada intervalo de tempo usado para construir os vetores característicos. Assim, se o tempo de desenvolvimento do projeto é dividido em doze intervalos, o Espaço das Classes é equivalente ao R^{12}. A representação de um vetor característico no Espaço das Classes é simples: cada célula do vetor está relacionada a uma dimensão no Espaço das Classes e o valor da célula representa o valor da coordenada nessa dimensão. Como o valor de uma célula de um vetor característico só pode ser zero ou um, todos os vetores são representados nos vértices de um hiper-cubo localizado no primeiro quadrante do Espaço das Classes. A Figura 3 apresenta este hiper-cubo em um espaço tridimensional.

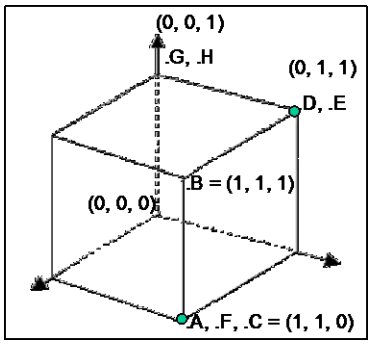

Figura 3. Um espaço de classes com intervalos de três tempos (R^3) e vetores característicos de algumas classes

Considere duas classes, D e E, que não foram modificadas durante o primeiro intervalo de tempo, mas foram no segundo e terceiro intervalos. De acordo com a representação sugerida, as duas classes deveriam ser representadas pelo mesmo vetor característico, localizado no canto superior direito do hiper-cubo da Figura 3. Por outro lado, considerando as classes D e A (a classe A foi modificada somente no primeiro e no segundo intervalos de tempo) observamos que eles estão em vértices opostos de uma das faces do hiper-cubo.

Para descobrir se uma distribuição de classes em pacotes está de acordo com o PCC, devemos criar uma medida de comparação entre duas classes em relação à ocorrência de mudanças concomitantes. Baseando-se na representação das classes no espaço geométrico, a distância entre duas classes pode ser calculada usando a distância Euclidiana. Esta é a distância entre dois pontos $P = (p_1, p_2 \ldots p_n)$ e $Q = (q_1, q_2 \ldots q_n)$ em um espaço n-dimensional, calculada pela Equação

(1). Considerando a Figura 3, a distância entre as classes D e E é zero, enquanto a distância entre D e A é $\sqrt{2}$.

$$\sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\ldots+(p_n-q_n)^2}=\sqrt{\sum_{i=1}^{n}(p_i-q_i)^2} \quad (1)$$

Outros estudos [15] [16] [17] têm usado formalmente uma analogia geométrica para calcular a distância conceitual entre elementos. Por exemplo, Dias-Neto e Travassos [17] usam a distância Euclidiana para apoiar a seleção de técnicas de teste para projetos de software, explorando a distância entre as características de um projeto de software e um conjunto de técnicas de teste baseadas em modelos, cada qual representada em um espaço de vetores.

Através do cálculo da distância Euclidiana entre cada par de classes, construímos uma matriz de distâncias onde cada linha e cada coluna representa uma classe e o valor de uma célula contém a distância entre as classes relacionadas à sua linha e à sua coluna. Esta matriz é simétrica e possui diagonal igual a zero (as células pertencentes à diagonal relacionam uma classe a si mesma; como a distância entre uma classe e ela mesma é zero, as células da diagonal da matriz são sempre zero), como mostrado na Figura 4.

	A	B	C	D	E	F	G	H
A	0	1	0	$\sqrt{2}$	$\sqrt{2}$	0	$\sqrt{3}$	$\sqrt{3}$
B	1	0	1	1	1	1	$\sqrt{2}$	$\sqrt{2}$
C	0	1	0	$\sqrt{2}$	$\sqrt{2}$	0	$\sqrt{3}$	$\sqrt{3}$
D	$\sqrt{2}$	1	$\sqrt{2}$	0	0	$\sqrt{2}$	1	1
E	$\sqrt{2}$	1	$\sqrt{2}$	0	0	$\sqrt{2}$	1	1
F	0	1	0	$\sqrt{2}$	$\sqrt{2}$	0	$\sqrt{3}$	$\sqrt{3}$
G	$\sqrt{3}$	$\sqrt{2}$	$\sqrt{3}$	1	1	$\sqrt{3}$	0	0
H	$\sqrt{3}$	$\sqrt{2}$	$\sqrt{3}$	1	1	$\sqrt{3}$	0	0

Figura 4 – Uma matriz de distância para oito classes

A matriz de distância apresenta uma avaliação por pares de quão freqüentemente as classes mudam nos mesmos intervalos de tempo. Assim, usamos esta matriz como entrada para uma técnica que distribui classes em pacotes de acordo com a freqüência de mudança conjunta. A próxima seção apresenta esta técnica.

IV. ORGANIZANDO CLASSES EM PACOTES

Dado um conjunto C de classes e um conjunto P_{ANTES} de pacotes representando a distribuição atual das classes nos pacotes (como expresso nos documentos de projeto e no código fonte do sistema), a técnica proposta sugere um conjunto P_{DEPOIS} de pacotes com uma distribuição de classes por pacotes que segue os direcionamentos propostos pelo PCC. A técnica pode ser formalmente descrita como abaixo:

f: (C, P_{ANTES}, R, $d_{INÍCIO}$, Δ, r_{min}) → P_{DEPOIS}

sujeito a,

$r_{min} \geq 1$,

$d_{INÍCIO} \geq \min(r.d), \forall\ r \in R$;

$d_{INÍCIO} \leq \max(r.d), \forall\ r \in R$;

onde,

C é o conjunto de classes que compõem o sistema;

P_{ANTES} é o conjunto de pacotes que compõem o sistema, de acordo com sua implementação corrente;

R é o conjunto de revisões de classes coletadas do *log* do sistema de controle de versão com o histórico de alterações;

$d_{INÍCIO}$ é a data a partir da qual a técnica irá considerar revisões $r \in R$;

Δ é o intervalo de tempo no qual as mudanças serão observadas (usado para construir os vetores característicos);

r_{min} é o número mínimo de revisões para uma classe ser considerada na técnica, e

P_{DEPOIS} é o conjunto de pacotes a ser utilizado pelo sistema depois da redistribuição de classes por pacotes.

A identificação de que classes devem fazer parte de cada pacote, de acordo com seu perfil de modificação conjunta, é um problema de agrupamento. O termo *Cluster Analysis*, apresentado pela primeira vez por [18], representa um conjunto de algoritmos de classificação que se destinam a organizar um conjunto de elementos em grupos, de modo que cada grupo contenha os elementos mais similares entre si e mais diferentes dos elementos pertencentes a outros grupos.

Existem várias abordagens para resolver problemas de agrupamento quando os elementos a serem agrupados são representados como vetores e os grupos que irão conter esses elementos podem ser descritos da mesma maneira. Selecionamos o algoritmo de *Cluster Analysis k-Means* [19] como base para a técnica de distribuição de classes em pacotes de acordo com o perfil de mudança das classes. O algoritmo de agrupamento *k-Means* é um método de *Cluster Analysis* que particiona "n" observações em "k" grupos, de forma que cada observação pertença ao grupo com a média mais próxima. O *k-Means* é um método de agrupamento não hierárquico, onde os grupos resultantes são independentes entre si ao invés de serem organizados em uma hierarquia.

O algoritmo *k-Means* pode ser resumido como: (i) selecionar o número de grupos desejado, conhecido como "k"; (ii) distribuir aleatoriamente os "n" elementos a serem classificados, representados como vetores, em "k" grupos; (iii) calcular o centróide de cada grupo, como a média dos vetores que representam os elementos pertencentes ao grupo; (iv) calcular a distância de cada elemento para o centróide de cada grupo; (v) transferir cada elemento para o grupo cujo centróide esteja mais próximo do vetor que representa o elemento; e (vi) repetir os passos 3-5 até que nenhum elemento seja transferido para outro grupo.

Uma limitação do algoritmo *k-Means* reside em sua dependência de uma definição prévia do número de grupos desejado. Assim, para aplicar esse algoritmo como parte da técnica de distribuição classes em pacotes, é necessário indicar o número de pacotes que serão formados de acordo com as propriedades do sistema sob análise. Para verificar as sugestões, por vezes discordantes, sobre o número aceitável de classes por pacote apresentadas pela literatura (ver seção II), organizamos uma pesquisa para identificar o tamanho médio de um pacote. Neste sentido, o código-fonte de vários softwares desenvolvidos segundo o modelo *Open Source* foi estudado (ver Tabela 1). Embora reconheçamos as limitações de usar apenas software de código aberto em estudos empíricos de Engenharia de Software, não tínhamos a intenção de abordar a questão profundamente, mas ter um direcionamento para interpretar os números fornecidos pela literatura. Os sistemas utilizados nesta análise foram escolhidos por conveniência, em função da disponibilidade de código-fonte, uso da mesma linguagem de programação (Java) e do mesmo sistema de controle de versões (CVS).

Após a análise de nove sistemas de código aberto (6 projetos grandes, 2 médios e 1 pequeno), foram obtidos os dados constantes na Tabela 1. Para cada sistema em análise, a tabela mostra o número médio de classes por pacote (MÉD C/P), o número de classes no menor pacote (MÍN C/P) e o número de classes no maior pacote (MÁX C/P) utilizado pelo sistema.

Tabela 1- Número de classes por pacote em nove sistemas de código aberto

Software	***MÉD C/P***	***MÍN C/P***	***MÁX C/P***
JDK6	16,8	1	216
JDK1.4	19,5	1	182
Eclipse	7,7	1	162
NetBeans	6,3	1	129
JBoss	6,2	1	78
Azureus	5,5	1	31
jVI	16,6	3	53
FMJ	4,6	1	88
OpenOffice	6,8	1	126
Média	10	-	-

Oito dos nove projetos de código aberto têm pacotes com uma única classe. O número médio de classes por pacote é cerca de 10, mas essa média é fortemente influenciada por três projetos: o Java Development Kit versão 6, que tem um pacote com 216 classes, uma versão anterior do Java Development Kit (versão 1.4), que tem um pacote com 182 classes e o editor de texto (jVI), cujo maior pacote engloba mais de 50% das classes do projeto. Se tais projetos são descartados em nossa análise, o número médio de classes por pacote é de aproximadamente 6, inferior à média apresentada na literatura [13] e próximo a mediana original.

Definimos o número de pacotes como um parâmetro, deixando para o arquiteto de software decidir quantos pacotes devem ser utilizados em um sistema. O número médio de classes por pacote em sistemas de código aberto e outras referências da literatura de Engenharia de Software (ver seção II) podem ser usados para apoiar esta decisão.

Em nossos experimentos, utilizamos o mesmo número de pacotes que o sistema possui em sua implementação atual. Cada pacote com pelo menos uma classe na composição hierárquica dos pacotes (onde um pacote pode conter outros sub-pacotes) é considerado pelo algoritmo. Além disso, ao invés de iniciar o algoritmo *k-Means* com uma distribuição aleatória de classes em pacotes, alteramos a segunda etapa do algoritmo para utilizar a atual distribuição de classes em pacotes de acordo com os documentos de projeto e o código-fonte do sistema.

Uma vez que todas as classes são representadas como vetores característicos em um espaço n-dimensional, o centróide de um pacote também é representado como um vetor característico, calculado de forma que cada uma de suas dimensões contenha o valor médio dos vetores característicos das classes residentes no pacote para aquela dimensão. Para identificar se uma classe deve permanecer no seu pacote ou mudar para outro, o algoritmo calcula a distância entre o vetor característico da classe e os vetores característicos de todos os pacotes. Em seguida, o algoritmo identifica o pacote mais próximo da classe e a transfere para ele. Depois de completar uma iteração do algoritmo *k-Means*, o centróide de cada pacote é recalculado, uma vez que as classes transferidas para outros pacotes podem afetar a sua posição.

Após a execução do algoritmo *k-Means* modificado em alguns sistemas de software, percebemos que, com certa freqüência, um grande número de classes convergiu para o mesmo pacote. Esta convergência produz *God Packages* [13] - um pequeno conjunto de pacotes que concentra a maioria das classes e, conseqüentemente, a maior parte da lógica da aplicação. *God Packages* são considerados uma má prática em projetos de software, pois lhes falta a capacidade de fornecer uma divisão compreensível das classes que implementam a aplicação, muitas vezes levando a projetos difíceis de entender e modificar.

Para evitar a formação de pacotes muito grandes e muito pequenos na técnica de distribuição de classe, adicionamos dois passos ao final do algoritmo *k-Means*. Estes passos mesclam pacotes que estão abaixo de um tamanho mínimo e quebram os pacotes muito grandes em dois ou mais.

O primeiro passo determina quais pacotes estão abaixo de um tamanho mínimo. Para identificar se um pacote é muito pequeno, usamos um parâmetro que representa o número mínimo aceitável de classes em um pacote. Nestes casos, as classes que compõem o pacote de pequeno porte são movidas para outro pacote e o antigo é descartado. Depois de distribuir as classes nos pacotes, qualquer pacote com menos classes do que o número indicado neste parâmetro é

mesclado com o pacote mais próximo disponível. Para selecionar o pacote mais próximo, a distância euclidiana entre o centróide do pacote a ser descartado e os centróides de todos os outros pacotes é calculada. Em nossos experimentos, utilizamos duas classes como o tamanho mínimo de um pacote.

A segunda etapa determina os pacotes que têm mais que um número máximo de classes. Para identificar se um pacote é muito grande, usamos um parâmetro que representa o número máximo aceitável de classes em um pacote. Pacotes com mais classes do que o indicado neste parâmetro devem ser quebrados em pacotes menores de acordo com o número ideal de classes por pacote. Este número permite calcular quantos novos pacotes devem ser criados para quebrar o pacote grande. O cálculo é feito através da divisão inteira do número de classes do pacote grande pelo número ideal de classes por pacote. Após distribuir as classes, pacotes com mais classes que o permitido são divididos em dois ou mais pacotes. Suas classes são distribuídas aleatoriamente entre os pacotes recém criados, de modo que nenhum deles concentre mais classes do que o estipulado no número ideal de classes por pacote. Em linha com a análise realizada sobre projetos de código aberto e na literatura [3][5][6], utilizamos, em nossos experimentos, 7 classes por pacote como o número ideal de classes por pacote e 20 classes por pacotes como o tamanho máximo dos pacotes.

Se os dois passos anteriores fundirem ou desmembrarem qualquer pacote, o número total de pacotes utilizados no algoritmo *k-Means* se torna diferente do que foi usado na execução inicial do algoritmo. Assim, o processo de agrupamento é executado com o novo número de pacotes para encontrar uma melhor distribuição das classes por pacotes de acordo com os princípios do PCC. No entanto, as tentativas de distribuir as classes sem formar pacotes demasiadamente pequenos ou muito grandes podem nunca ter sucesso: o algoritmo *k-Means* pode concentrar classes em torno de alguns pacotes não importa quantos novos pacotes sejam criados. Para evitar um algoritmo que nunca encerre sua execução sob certas situações, os passos que verificam o tamanho dos pacotes são repetidos um número limitado de vezes (indicado em um parâmetro). Em nossos experimentos, utilizamos no máximo 10 iterações, executadas em seqüência e de forma automática.

V. AVALIANDO A ABORDAGEM PROPOSTA

A fim de avaliar a técnica proposta na seção IV, foi realizado um estudo experimental com oito sistemas de software de código aberto. O estudo buscou responder à seguinte questão: *a estrutura de um sistema de software é melhorada se for reformulada de acordo com a técnica de distribuição de classes por pacotes baseada no Princípio Common-Closure?*

O objetivo deste estudo foi verificar a viabilidade de utilizar a técnica proposta para reorganizar as classes que compõem o software orientado a objetos, a fim de melhorar a adequação de seu projeto ao PCC. Avaliamos esta melhoria pela coleta de dados sobre métricas de projeto de software antes e depois de aplicar a técnica de distribuição de classes por pacotes. Foi realizado um experimento sob a perspectiva do pesquisador, avaliando a viabilidade da técnica proposta a fim de apoiar o processo de melhoria contínua de nossa pesquisa. O estudo foi realizado como um teste individual *in silico* [21] sobre uma série de objetos, cada um sendo um projeto de software no qual tivemos acesso ao histórico de mudanças a partir do sistema de controle de versão.

Devido à disponibilidade e visando focar o estudo, restringimos os objetos analisados a softwares desenvolvidos usando o modelo de software livre, tendo Java como linguagem de programação e CVS como sistema de controle de versão. Foram selecionadas oito aplicações no SourceForge e repositórios do projeto Eclipse, baixado o código fonte e os *logs* com histórico de alterações. Os sistemas selecionados são apresentados na Tabela 2.

Tabela 2- Breve descrição das aplicações sob análise

Software	*Breve Descrição*
Azureus	Cliente para compartilhamento de arquivos P2P usando o protocolo *bit torrent*.
Eclipse ANT	Ferramenta de *build* construída em Java e apresentada como um módulo do Eclipse.
Jena	Framework Java para construção de aplicações para a Web semântica.
JMule	Cliente para compartilhamento de arquivos escrito em Java para redes eDonkey2000.
JUnit	Framework Java que permite a criação de classes para testes unitários.
JVI	Implementação Java do clássico editor de texto VI.
Poor Man CMS	SGC que é executado como uma aplicação Java e gera páginas HTML estáticas.
Sweet Home 3D	Aplicação para projeto de interiores, que permite a alocação de móveis em uma casa desenhada em 2D, com *preview* 3D.

Tabela 3 - Características das aplicações sob análise

Software	*Classes*	*Pacotes*	*Intervalos*
Azureus	349	59	646
Eclipse ANT	37	4	277
Jena	1413	80	59
JMule	616	89	148
JUnit	449	51	60
JVI	143	5	231
Poor Man CMS	49	15	17
Sweet Home 3D	299	10	350

No que diz respeito ao seu projeto, estas aplicações têm as características apresentadas na Tabela 3. O número de classes e pacotes representa o número dos respectivos elementos, na versão original do software, como quando

baixado do SourceForge ou Eclipse. O número de intervalos representa o número de intervalos de tempo em que o *log* do histórico de mudanças da aplicação foi dividido. Um pequeno número de intervalos geralmente representa uma aplicação recente, com histórico de alterações curto. Por outro lado, um grande número de intervalos representa uma aplicação mais antiga, com ciclo de desenvolvimento mais longo e que sofreu mais alterações ao longo do tempo.

A hipótese nula para o estudo determina que o uso da técnica proposta de distribuição de classes em pacotes (e, conseqüentemente, a aplicação do princípio de projeto PCC) não prevê melhorias significativas na estrutura de um sistema. Por outro lado, a hipótese alternativa indica que há melhoria ou perda estrutural no software após a aplicação da técnica proposta. Esta melhoria foi avaliada de acordo com um conjunto de métricas de qualidade de projeto de software. Assim, a hipótese nula indica que não haverá diferença significativa quando essas métricas forem colhidas antes e depois da redistribuição de classes em pacotes.

Para avaliar as hipóteses propostas, escolhemos duas métricas de acoplamento de software.

- *Afferent coupling* (AFF): o acoplamento de entrada de um pacote P é uma contagem de classes que dependem de classes pertencentes a P para realizar o seu contrato. Somente as classes localizadas fora do pacote P são contadas;
- *Efferent coupling* (EFF): o acoplamento de saída de um pacote P é o número de classes que pertencem a P, mas que dependem de classes de outros pacotes para cumprir seu contrato.

Já que um bom projeto geralmente apresenta baixo acoplamento, a hipótese nula pode ser declarada como apresentado a seguir (ANT = antes de aplicar a técnica de redistribuição de classes; DEP = depois da aplicação da técnica proposta).

H_0: $\mu\ AFF_{ANT} = \mu\ AFF_{DEP} \wedge \mu\ EFF_{ANT} = \mu\ EFF_{DEP}$

A hipótese alternativa é definida a seguir, como um complemento de H_0.

H_A: $\mu\ AFF_{ANT} \neq \mu\ AFF_{DEP} \vee \mu\ EFF_{ANT} \neq \mu\ EFF_{DEP}$

O estudo experimental foi decomposto nas seguintes etapas: (i) o código fonte e o *log* do histórico de alterações sofridas pela aplicação foram recuperados; (ii) as aplicações foram medidas, registrando-se o número de classes, pacotes, acoplamento de entrada e saída; (iii) as aplicações foram submetidas à técnica proposta, gerando uma sugestão de distribuição de classes por pacotes; (iv) as aplicações foram refatoradas para atender à distribuição proposta; (v) as aplicações refatoradas foram novamente medidas; e (vi) os valores coletados para as métricas de acoplamento, antes e depois da refatoração foram comparados.

Mantivemos um pequeno número de sistemas selecionados para o estudo porque tivemos de refatorar manualmente cada sistema para replicar o mapeamento das classes para pacotes proposto pela técnica de distribuição. Mesmo utilizando ambientes de desenvolvimento integrados (IDE) modernos, tais refatorações foram custosas em termos de horas de trabalho.

As tabelas 4 e 5 apresentam a média e o desvio-padrão para o acoplamento de entrada e de saída das aplicações selecionadas para o estudo experimental, antes e depois da refatoração.

Tabela 4 – Média de valores para acoplamento de entrada e saída

Software	AFF_{ANT}	AFF_{DEP}	EFF_{ANT}	EFF_{DEP}
Azureus	9.3	16.1	4.5	4.5
Eclipse ANT	3.3	4.0	7.3	4.7
Jena	38.6	70.8	13.7	37.0
JMule	13.9	21.8	4.5	5.3
JUnit	10.6	16.3	4.0	5.2
JVI	7.8	15.2	8.6	4.8
Poor Man CMS	10.7	12.0	3.2	4.0
Sweet Home 3D	27.9	26.0	12.4	5.0

Tabela 5 – Desvio padrão para acoplamento de entrada e saída

Software	AFF_{BEF}	AFF_{AFT}	EFF_{BEF}	EFF_{AFT}
Azureus	12.5	11.9	5.1	2.5
Eclipse ANT	3.3	2.6	4.3	4.3
Jena	84.8	121.1	12.6	139.8
JMule	23.2	22.9	4.3	5.1
JUnit	20.5	22.6	2.9	8.5
JVI	6.6	7.2	8.9	2.9
Poor Man CMS	8.3	7.5	1.7	3.9
Sweet Home 3D	32.7	24.3	11.0	2.4

Submetendo estes valores a um teste de Mann-Whitney bicaudal usando $\alpha = 90\%$, a hipótese nula de igualdade foi rejeitada para as duas métricas de acoplamento, tanto para o desvio-padrão ou média. O teste não paramétrico de Mann-Whitney foi escolhido para a análise por falta de evidências que caracterizem a distribuição dos valores observados. Uma vez que os valores médios para acoplamento são maiores após a aplicação da técnica proposta (exceto para o Sweet Home 3D e acoplamento de saída no Eclipse ANT e jVI), temos um indício de que a utilização do princípio de projeto PCC não melhora o acoplamento dos sistemas.

Agrupamos também os sistemas de acordo com o número de classes, o número de intervalos de tempo no *log* de mudanças e a relação entre essas medidas. Selecionamos as aplicações de cada grupo de modo que a soma dos desvios-padrão para cada grupo seja mínima, conforme apresentado em [20]. Para cada grupo, calculamos a variação de AFF e EFF como um percentual destes valores antes e depois da refatoração das aplicações, de acordo com as sugestões da técnica proposta. As tabelas 6, 7 e 8 apresentam os diferentes grupos.

Tabela 6 – Variação de acordo com o número de classes

Grupos de Software (número de classes)	*MÉD classes*	*AFF Variação*	*EFF Variação*
Eclipse ANT, Poor Man CMS, JVI	77	+46%	-18%
Azureus, Jena, JMule, JUnit, Sweet Home 3D	625	+52%	+32%

Tabela 7 – Variação de acordo com o número de intervalos de tempo

Grupos de Software (número de intervalos)	*MÉD intervalos*	*AFF Variação*	*EFF Variação*
Poor Man CMS, Jena, JUnit	46	+50%	+75%
Eclipse ANT,Azureus, JMule, JVI, Sweet Home 3D	330	+48%	-24%

Tabela 8 – Variação de acordo com a taxa classe/intervalo

Grupos de Software (taxa classe/intervalo)	*MÉD C/I taxa*	*AFF Variação*	*EFF Variação*
Eclipse ANT, JVI, Azureus	43%	+63%	-26%
Poor Man CMS, Jena, JUnit, JMule, Sweet Home 3D	787%	+40%	+37%

Observou-se pouca variação para a AFF em todos os grupos: o acoplamento de entrada parece aumentar sempre. Por outro lado, a técnica foi capaz de encontrar uma distribuição de classes que diminui o acoplamento de saída, desde que o histórico de mudanças da aplicação possa ser dividido em mais intervalos de tempo e/ ou exista um pequeno número de classes para organizar entre os pacotes. Essa conclusão era esperada, pois a relação entre o número de classes e o número de intervalos de tempo é uma medida de quanta "informação sobre mudanças" está disponível para ser utilizada pela técnica. Observamos uma forte correlação (95%) entre a relação classes/intervalos e o aumento no acoplamento de saída. Uma vez que o acoplamento de saída é uma medida mais próxima da independência do pacote do que o acoplamento de entrada, observou-se que a técnica de redistribuição de classes pode ser útil quando existir informação suficiente sobre as alterações sofridas pela aplicação – relação pequena entre o número de classes e de intervalos.

Assim, sugerimos que o *log* com o histórico de alterações deve ter o dobro de intervalos de tempo que o número de classes que compõem a aplicação. Esta relação não é facilmente observável em projetos de software, uma vez que um sistema composto por 1.000 classes exigiria 2.000 intervalos de tempo diários (ou quase 6 anos de histórico). Esta limitação fará com que a técnica proposta dificilmente seja útil para sistemas que estejam em desenvolvimento, embora ela possa ajudar a melhorar a estrutura de grandes sistemas durante a sua manutenção.

VI. LIMITAÇÕES DA ABORDAGEM PROPOSTA

Primeiramente, a amostra de aplicações de software selecionada para análise no estudo experimental é relativamente pequena. Isto impõe uma ameaça à conclusão, uma vez que a força do teste estatístico (Mann-Whitney) é limitada pela pequena amostra. Por outro lado, selecionamos uma amostra focada, limitando as aplicações a programas Java, de código aberto, e utilizando um sistema de controle de versão específico (CVS). Assim, acreditamos que nossas conclusões valerão para tal cenário, especialmente para outros projetos de software criados ou mantidos usando uma estratégia *Open Source*.

Além disso, a técnica é baseada em uma suposição forte de que as submissões para o repositório central de um sistema de controle de versão refletem as alterações reais impostas pelos desenvolvedores às classes compreendendo um software. Esta suposição é forte porque é influenciada por fatores não relacionados à perspectiva de projeto de software, tais como o comportamento do desenvolvedor e políticas corporativas. Por exemplo, pode-se argumentar que os desenvolvedores alteram as classes com freqüência, mas submetem essas alterações para o repositório de controle de versão apenas esporadicamente. Por outro lado, outros desenvolvedores podem submeter com freqüência para permitir aos demais seguir seus avanços e construir sobre suas implementações feitas parcialmente.

Esses diferentes padrões de comportamento podem influenciar nossa capacidade de identificar as classes alteradas no mesmo intervalo de tempo, afetando a percepção de alterações concomitantes subjacente à nossa técnica. Isto impõe uma ameaça à validade de construção uma vez que nosso instrumento (*commit*) pode não representar precisamente a teoria em estudo (mudanças e o PCC). No entanto, acreditamos que não há alternativa automatizada para o histórico de alterações sofridas por artefatos de software fornecido pelos sistemas de controle de versão. Por outro lado o registro manual destas alterações não é um procedimento prático para grandes projetos de software ou projetos geograficamente distribuídos. Assim, acreditamos que a hipótese é sólida o suficiente para suportar a conclusão.

Outra limitação é que ela não está contemplando o aspecto hierárquico da estrutura pacote. A técnica gera um conjunto de pacotes plano, enquanto a estrutura típica do código-fonte é uma hierarquia, com pacotes grandes representando toda a aplicação, contendo pacotes menores ou subsistemas. Pretendemos abordar este aspecto, no futuro, possivelmente complementando os resultados obtidos pelo *k-Means* com um algoritmo de agrupamento hierárquico. Tais algoritmos utilizam a distância entre os pares de grupos (no nosso caso, centróides do pacote) para organizá-los em clusters, criando assim uma hierarquia, onde grupos com médias similares são apresentados como partes de grupos maiores.

A técnica proposta não está levando em consideração os diferentes desenvolvedores que podem ter trabalhado de forma concorrente durante o desenvolvimento de software. Assim, em um mesmo momento, diversas alterações distintas poderiam estar em curso, quebrando a premissa de que classes alteradas nos mesmos instantes de tempo representam o mesmo eixo de mudança. Pretendemos realizar novos experimentos no futuro, filtrando os espaços de mudança de acordo com os desenvolvedores que realizaram estas mudanças e separando, assim, diferentes perfis de alteração do sistema.

A ferramenta utilizada, em sua versão atual, exige que o usuário efetue os *refactorings* sugeridos de forma manual. Esta provou ser a etapa que mais consome tempo no processo utilizado no estudo experimental. A ferramenta está sendo adaptada para automatizar a coleta das métricas sem a necessidade de *refactoring* em si.

VII. CONCLUSÃO

Neste trabalho apresentamos uma técnica para distribuir as classes, incluídas num sistema de software orientado a objeto, em pacotes, de acordo com o princípio de projeto de pacote *Common-Closure*. Este princípio dita que as classes que mudam juntas devem pertencer ao mesmo pacote, e a técnica proposta utiliza informações do *log* de um sistema de controle de versão com o histórico de alterações como uma *proxy* para as alterações sofridas pelas classes.

Ao aplicar a técnica proposta em oito aplicações de código aberto, discutimos se o uso do PCC poderia melhorar seus projetos em termos de acoplamento. Observou-se que uma relação pequena entre o número de classes e o número de intervalos de tempo em que foram observadas alterações é fundamental para reduzir o acoplamento. Assim, a técnica proposta pode ser útil para diminuir o esforço de manutenção em sistemas de software de grande porte.

AGRADECIMENTOS

Os autores gostariam de agradecer e reconhecer o apoio financeiro concedido a este projeto pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e pela Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ).

REFERÊNCIAS

[1] M.M. Lehman, J.F. Ramil, P.D. Wernick, et al., “Metrics and Laws of Software Evolution – The Nineties View”. In: Proceedings of the Fourth Intl. Software Metrics Symposium (Metrics’97), pp.20, 1997, Albuquerque, NM.

[2] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices. New Jersey, Prentice Hall, 2002.

[3] H. Melton, E. Tempero, “The CRSS Metric for Package Design Quality”. In: Proceedings of the Thirtieth Australasian Conference on Computer Science, pp. 201-210, 2007, Ballarat, Victoria, Australia.

[4] S. Gibbs, D. Tsichritzis, E. Casais, et al., “Class Management for Software Communities”, Communications of the ACM, v. 33, n. 9, pp.90-103, September 1990, New York, USA

[5] J. Lakos, Large-scale C++ software design, Redwood City, Addison Wesley Longman Publishing, CA, USA, 1996.

[6] J. Bay, The ThoughtWorks Anthology – Essays on Software Technology and Innovation, The Pragmatic Bookshelf, Dallas, USA, 2008.

[7] B. Meyer, Object-Oriented Software Construction, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, USA, 1997.

[8] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufmann, 2002.

[9] A. Riel, Object-Oriented Design Heuristics. Addison Wesley, Boston MA, 1996.

[10] K. J. Lieberherr, Adaptative Object Oriented Software: The Demeter Method, PWS Publishing, 1996.

[11] L. Martin, A. Giesl, J. Martin, “Dynamic Component Program Visualization”, In Proceedings of the WCRE 2002, 2002.

[12] S. McConnell, Code Complete, Second Edition, Microsoft Press, 2004

[13] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer Verlag, Berlin Heidelberg, 2006.

[14] J. Mayo, 2000, “The C# Station: Namespaces”. available at http://www.csharp-station.com/Tutorials/Lesson06.aspx, last accessed in March, 20th, 2010

[15] C. Moneta, G. Vernazza, R. Zunino, “A Vectorial Definition of Conceptual Distance for Prototype Acquisition and Refinement”. Technical Report TUM-I9019, Technical University of Munich, 1990.

[16] T. Asada, R.F. Swonger, N. Bounds et al., “The Quantified Design Space: A Tool for the Quantitative Analysis of Design”, SEI Technical Report CMU/SEI-92-TR213, Carnegie Mellon University, 1992.

[17] A. C., Dias-Neto, and G. H. Travassos, "Model-based testing approaches selection for software projects", Information and Software Technology. Vol. 51, Issue 11, pp. 1487-1504, November 2009

[18] Tryon, R. C. Cluster analysis. Ann Arbor: Edwards Brothers, 1939.

[19] S. P. Lloyd, "Least squares quantization in PCM". IEEE Transactions on Information Theory 28 (2): pp 129–137, 1982.

[20] H. R. Costa, M. O. Barros, G. H. Travassos, “Evaluating software project portfolio risks”. Journal of Systems and Software 80(1): 16-31 (2007)

[21] G. H. Travassos, M. O. Barros, “Contributions of in virtuo and in silico experiments for the future of empirical studies in Software Engineering”. IN: 2nd Workshop Series on Empirical Software Engineering: The Future of Empirical Studies in Software Engineering, Proceedings of the WSESE 2003, 2003, Rome