UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CHANGEFINDER: AVALIANDO OS IMPACTOS DO USO DO PRINCÍPIO DE PROJETO DE PACOTES COMMON-CLOSURE NA MANUTENÇÃO DE UM SOFTWARE ORIENTADO A OBJETOS

Marcelo de França Costa

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2010

CHANGEFINDER: AVALIANDO OS IMPACTOS DO USO DO PRINCÍPIO DE PROJETO DE PACOTES COMMON-CLOSURE NA MANUTENÇÃO DE UM SOFTWARE ORIENTADO A OBJETOS

Marcelo de França Costa

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

Márcio de Oliveira Barros, D.Sc. – UNIRIO

Renata Mendes de Araujo, D.Sc. – UNIRIO

Alexandre Albino Andreatta, D.Sc. – UNIRIO

Leonardo Gresta Paulino Murta, D.Sc. – UFF

RIO DE JANEIRO, RJ – BRASIL

JULHO DE 2010

Costa, Marcelo de França.
C837 Changefinder – avaliando os impactos do uso do princípio de projeto de pacotes Common-Closure na manutenção de um software orientado a objetos / Marcelo de França Costa, 2010.
87f.

Orientador: Márcio Oliveira Barros.
Dissertação (Mestrado em Informática) – Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2010.

1. Sistemas de pacotes. 2. Evolução de software. 3. Métricas de software. 4. Software – Manutenção. I. Barros, Márcio Oliveira. II. Universidade Federal do Estado do Rio de Janeiro (2003-). Centro de Ciências Exatas e Tecnologia. Curso de Mestrado em Informática. III. Título.

CDD – 005.5

À minha querida amiga Danielle.

Aos meus pais.

Aos meus amigos e aos meus alunos.

Agradecimentos

A Deus, por todas as oportunidades a mim concedidas durante minha vida, dando-me perseverança, resiliência e iluminação para alcançar mais este objetivo.

Aos meus pais Darcy e Fátima, pelas cobranças (incentivo), apoio, amizade e amor incondicional dedicados durante todas as etapas da minha vida.

Ao meu orientador Prof. Márcio Barros, pelo voto de confiança, por acreditar no meu potencial, pela compreensão ao longo destes anos, por todas as dicas valiosas e direcionamento durante a realização deste trabalho.

Ao meu ex-coordenador José Wagner e à Value Team (Relacional Consultoria), pela concordância e apoio para realizar o curso de mestrado.

Aos demais professores do Programa de Pós Graduação em Informática da UNIRIO, pelo convívio, pelos ensinamentos e pelas críticas, em especial aos professores Renata Araujo e Alexandre Andreatta que possibilitaram o surgimento de tantas ideias utilizadas neste trabalho.

Aos funcionários da secretaria, cordiais e atenciosos, por sua paciência e dedicação aos nossos problemas administrativos e acadêmicos.

E por último, mas não por menos, aos amigos João Paulo e Vitor Braga, pela amizade e ajuda gratuitas a mim ofertadas logo no início desta caminhada.

Resumo da Dissertação apresentada ao PPGI/UNIRIO como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

CHANGEFINDER – AVALIANDO OS IMPACTOS DO USO DO PRINCÍPIO DE PROJETO DE PACOTES COMMON-CLOSURE NA MANUTENÇÃO DE UM SOFTWARE ORIENTADO A OBJETOS.

Marcelo de França Costa

Julho/2010

Orientador: Márcio de Oliveira Barros

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. No entanto, pouco se sabe sobre a real utilidade destes princípios e sua utilização pelos desenvolvedores de software. Neste trabalho, é proposta uma técnica para organizar as classes que compõem um sistema em pacotes de acordo com o princípio de projeto de pacotes Common-Closure. Apresenta-se o resultado de um estudo experimental para verificar se a adoção deste princípio melhora um conjunto de métricas de projeto de software. Conclui-se que a técnica proposta requer mais informação do que geralmente está disponível durante o desenvolvimento do software, mas pode apoiar a manutenção de projetos de software de larga escala.

Abstract of Dissertation presented to PPGI/UNIRIO as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

CHANGEFINDER – EVALUATING THE IMPACTS OF THE COMMON-CLOSURE PACKAGE DESIGN PRINCIPLE IN OBJECT ORIENTED SOFTWARE MAINTENANCE.

Marcelo de França Costa

July/2010

Advisor: Márcio de Oliveira Barros

Software systems are evolvable constructs that must be constantly changed to remain useful. However, the maintenance effort that is required to support this evolution is usually huge, and grows as the system ages and is changed in less than controlled ways. Software design principles propose ways to organize the basic components of software systems in order to accommodate change and reduce the maintenance effort. However, little is known about whether software practitioners effectively use such principles. In this work, the usage of the Common-Closure package design principle is addressed, proposing a strategy to organize the classes composing a system accordingly to this principle. It is also presented the results of an experimental evaluation that verifies if the application of the Common-Closure principle improves a set of software design metrics. It is concluded that the proposed technique requires more information than is usually available during the development of software, but can support the maintenance of large scale software projects.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1
INTRODUÇÃO

1.1. Motivação

Sistemas de software precisam evoluir ou as empresas que os desenvolveram correm o risco de perder uma fatia do mercado para seus concorrentes (LEHMAN *et al.* 1997). Entretanto, manter tais sistemas é uma tarefa difícil, complexa e que consome muito tempo e esforço. Adicionar novas funcionalidades, suportar novos equipamentos e plataformas, otimizar o sistema e corrigir defeitos se tornam atividades mais difíceis à medida que o sistema envelhece e cresce – *Software Entropy* (JACOBSON *et al.* 1992).

O paradigma da Orientação a Objetos (OO) procura facilitar as tarefas de desenvolvimento, através do reuso, bem como de manutenção, através de módulos de código mais coesos e, idealmente, menos acoplados. Apesar da contemporaneidade do software OO em relação ao código procedural, muitas aplicações OO comerciais já mostram os mesmos sinais de envelhecimento que o código legado procedural (SARKAR *et al.* 2008). Assim, um sistema que talvez tenha começado com uma arquitetura bem modularizada pode ter decaído e se transformado em um sistema de módulos interdependentes, que são difíceis de manter e estender de forma individual.

Um dos primeiros passos para se manter um sistema OO é detectar falhas de projeto (SALEHIE *et al.* 2006) que podem causar problemas para a futura evolução e manutenção. Os princípios de projeto OO são guias que devem ser utilizadas no início do ciclo de vida de um projeto, mas que igualmente podem ser aplicadas a qualquer momento, na forma de *refactorings*. Tais intervenções têm como objetivo minorar a entropia do software, ou seja, corrigir falhas ou melhorar sua estrutura, almejando facilitar futuras modificações.

Em geral, os princípios de projeto OO se dividem em princípios baseados em classes – como o *Open Closed* (MEYER 1997), a Substituição de Liskov (LISKOV e WING 1994) e a Lei de Demeter (LIEBERHERR *et al.* 1988) – e princípios baseados

em pacotes – como o Principio *Common-Closure* (MARTIN 2002) e o Princípio da Equivalência do Reuso e Liberação (MARTIN 2002). Softwares OO de larga escala são estruturados em Pacotes de Classes e os princípios da área de Projeto de Pacotes (ou *Package Design*) têm como objetivo orientar a organização de tais aplicações (MARTIN 2002). Apesar de sua notável importância, pouco se fala da real adoção de tais princípios ou do efetivo ganho de qualidade no software produzido.

1.2. Definição do Problema

Apesar de existirem sistemas de software grandes, complexos e ubíquos, pouco se sabe sobre as estruturas internas dos sistemas em produção (BAXTER *et al.* 2006). Uma grande quantidade de pesquisas tem focado em como o software deve ser escrito, ou seja, como seus componentes devem ser estruturados. Muitas regras, metodologias, notações, padrões e normas para projetar e programar tais sistemas têm sido produzidas (GAMMA *et al.* 1995, KRUCHTEN 2000, OMG 2004). Porém, pouco é conhecido sobre as estruturas de software de larga escala que existem no mundo real.

Com as metodologias, notações e outros princípios ou heurísticas que foram desenvolvidos, deveria ser possível constatar algo sobre o software resultante, assumindo que tais princípios sejam seguidos. Assim, a questão que vem à tona é determinar se as recomendações oferecidas têm sido adotadas ou não.

Existe alguma evidência de que o que é normalmente recomendado não é seguido. Por exemplo, muitos autores advertem quanto à criação de ciclos de dependência no software. Porém, trabalhos constataram que programadores não somente introduzem ciclos com certa regularidade, mas estes são muitas vezes grandes, contemplando vários módulos (BAXTER *et al.* 2006). Se uma determinada prática é recomendada, porém não é encontrada em softwares produzidos, isto pode indicar que a mesma não foi aceita pela comunidade acadêmica e técnica. A rejeição pode ter ocorrido por desconhecimento, falta de divulgação ou pela ausência de evidências de que a utilização da prática traga ganhos substanciais. Em se tratando de software OO, as recomendações focam basicamente em seus dois principais elementos estruturais: classes e pacotes.

A programação orientada a objetos já foi descrita como uma “tecnologia de empacotamento” (COX 1986). Empacotamento de classes se refere à questão de como representar a classe de um objeto de forma que a informação (estrutura e comportamento) necessária para usar tal classe seja facilmente localizada e

incorporada dentro de uma aplicação (GIBBS *et al.* 1990). As recomendações deste nível geralmente focam em como projetar e construir tais classes.

Já os pacotes estão ligados à organização das classes. Se o empacotamento da classe lida com a representação de classes individuais, a organização de classes por outro lado lida com o relacionamento e as dependências que ocorrem em coleções de classes (GIBBS *et al.* 1990). As recomendações desta granularidade focam em como projetar ou definir critérios a serem utilizados para constituir agrupamentos de classes.

Ao se analisar sistemas de softwares de larga escala, compostos de milhares ou milhões de linhas de código, torna-se imperativo o uso de estruturas de mais alto nível, já que classes possuem uma granularidade pequena. Considerando, então, um nível de granularidade maior (pacotes), pode-se dizer que não existem muitos trabalhos divulgando princípios de projeto correlatos ou trabalhos verificando se tais princípios estão sendo adotados nas comunidades de desenvolvimento de software. Também não existem muitos trabalhos que busquem identificar se os mesmos princípios conseguem efetivamente produzir software de melhor qualidade, como prometido.

Portanto, para buscar indícios de que uma determinada recomendação vem sendo utilizada e avaliar se a mesma realmente é capaz de produzir software com substanciais ganhos sob determinados aspectos de qualidade, é necessário um trabalho de análise do software em si. Esta análise, especializada para determinado princípio, deve procurar indícios da utilização da recomendação (por exemplo, um padrão de projeto) em softwares que se encontram em produção, bem como identificar características de qualidade do próprio software em comparação com outros – se valendo, para tal, de métricas de qualidade.

1.3. Objetivo

Com o objetivo de permitir uma análise que identifique se os princípios de projeto estão sendo seguidos, trabalhos realizados muitas vezes se valem de heurísticas que buscam detectar falhas de projeto (SALEHI *et al.* 2006). Tais trabalhos geralmente estão baseados em métricas (SARKAR *et al.* 2008) ou propõem novas métricas (MELTON e TEMPERO 2007), que também servem de indicativo da presença ou não do princípio de projeto. Também existem trabalhos que se baseiam em modelos ou *frameworks* que buscam identificar padrões de convergência na evolução do software em si (CAI e HUYNH 2007, BAXTER *et al.* 2006). Outro grupo de trabalhos oferece

soluções que visam auxiliar a implementação das boas práticas (WERNER *et al.* 2008, GIBBS *et al.* 1990).

Este trabalho está centrado na análise de sistemas de softwares que utilizem o paradigma de desenvolvimento da orientação a objetos. No contexto da orientação a objetos, o nível de granularidade de pacotes é preferível a classes quando se deseja um nível de abstração mais alto. Em se tratando de Padrões de Projeto, a área de Projeto de Pacotes é a que traz recomendações para tal nível de granularidade. E dentro desta área, este trabalho se propõe a analisar o Princípio *Common-Closure* (MARTIN 2002).

O princípio de projeto objeto de análise desta dissertação, como será visto em mais detalhes no capítulo 2, está fortemente ligado à construção/evolução do software em si. Sendo assim, e seguindo a estratégia de trabalhos recentes (MELTON e TEMPERO 2007, BAXTER *et al.* 2006), um repositório de código-fonte é utilizado como dados históricos de entrada para o processo de análise que se seguirá. Apesar de alguns estudos publicados sobre a evolução do software terem sido executados em sistemas desenvolvidos "*in house*" (dentro de uma única companhia) usando desenvolvimento e técnicas de gerenciamento tradicionais (GALL *et al.* 1997, KEMERER e SLAUGHTER 1999, LEHMAN *et al.* 1997, TURSKI 1996), este trabalho utiliza para seus experimentos softwares de código livre, disponíveis em repositórios públicos, como o SourceForge.

Como já foi mencionado em trabalhos anteriores, existem cuidados a serem tomados quando da utilização de código-livre para experimentos (HOWISON e CROWSTON 2004). Sendo assim, serão utilizados softwares considerados mais populares, ou seja, softwares com grande número de desenvolvedores ativos. Para efeito deste trabalho, e tendo como exemplo um trabalho recente (BAXTER *et al.* 2006), serão considerados apenas os softwares construídos utilizando a linguagem de programação Java.

Os objetivos deste trabalho podem ser descritos como: (a) buscar indícios da utilização do princípio de projetos *Common-Closure* em alguns sistemas de software; (b) propor um mecanismo que facilite a adequação de um sistema de software ao princípio, na forma de sugestões de *refactorings*; e (c) utilizando métricas de qualidade aceitas pela comunidade, tentar validar o princípio – buscando indícios de substanciais melhoras na qualidade do software caso ele seja modificado para se adequar ao princípio.

Para realização deste trabalho, foi construído um aplicativo nomeado *ChangeFinder* que operacionaliza o trabalho de análise. A vantagem de se utilizar este aplicativo é que o esforço para adaptação do software é reduzido, posto que o produto

final da análise é o software já reorganizado (código-fonte) segundo o princípio[1]. Também é produto final da análise a diferença entre as métricas de qualidade antes e após a aplicação do princípio através da técnica proposta. Os resultados obtidos com a utilização deste aplicativo são descritos no capítulo 4. A estrutura geral deste documento é descrita a seguir.

1.4. Estrutura da Dissertação

Este documento está dividido em cinco capítulos, como descrito a seguir: o capítulo 1 contém a introdução ao tema. Nesta, é apresentada a motivação para a realização deste trabalho, a contextualização do problema de modo amplo, os principais objetivos e a delimitação do escopo de solução. No capítulo 2 são apresentados trabalhos relacionados a princípios de projeto da área de Projeto de Pacotes, além do princípio em si que motivou este trabalho. No capítulo 3 é apresentada a técnica desenvolvida para detectar tanto a utilização do princípio, como sugerir *refactorings* de acordo com o mesmo. No capítulo 4 são descritos os resultados obtidos (utilizando os sistemas selecionados) através do uso do aplicativo construído – o aplicativo *ChangeFinder* também é descrito neste capítulo. No capítulo 5 a dissertação é encerrada, apresentando-se as conclusões baseadas nos resultados obtidos (descritos no capítulo 4), e algumas sugestões de trabalhos futuros são oferecidas.

[1] Na versão atual do aplicativo ChangeFinder, o sistema de software não é automaticamente reorganizado, mas apenas é listada a nova estrutura sugerida (classes por pacotes).

CAPÍTULO 2
CLASSES E PACOTES

2.1. Considerações Iniciais

O desenvolvimento de grandes sistemas de software é uma atividade complexa. Tais sistemas são compostos por centenas ou milhares de pequenos elementos (por exemplo, classes no paradigma orientado a objetos), que precisam ser combinados de forma a compor um sistema exeqüível. Estes mesmos sistemas tendem a evoluir em resposta às correções de defeitos (*bugs*) e adições de funcionalidades – ambas inerentes à natureza do software. Na medida em que as aplicações crescem em tamanho e complexidade, se torna necessária uma forma de organização de mais alto nível. Classes, embora sejam convenientes para organizar pequenas aplicações, são muito granulares para sistemas maiores. Pacotes (*namespaces*) são as estruturas, maiores que as classes, utilizadas para organizar grandes aplicações, de forma a obtermos uma visão de mais alto nível do sistema e organizarmos o desenvolvimento de seus componentes (ou subsistemas).

Um dos desafios ao se agrupar classes em subsistemas é que para qualquer conjunto de classes existem muitas formas possíveis de particioná-las (MARTIN 2002). A escolha dos particionamentos é fortemente influenciada pelas classes que compõem o sistema, uma vez que são os relacionamentos entre as classes que geram os relacionamentos entre os pacotes – *clusters*, grupos, diretórios ou *namespaces* – em um dado particionamento. Por conseqüência, relacionamentos entre tais pacotes podem ser alterados movendo-se classes entre partições ou alterando-se o código-fonte destas classes de forma a interromper seus relacionamentos com outras classes.

Ao projetar sistemas de software, é importante organizar seus componentes de uma forma que melhore sua manutenibilidade e reusabilidade. A subárea da disciplina de projeto de software conhecida como Projeto de Pacotes, discutida na seção 2.3, é a área preocupada em determinar a melhor maneira de se organizar as classes de um sistema em subsistemas (MELTON e TEMPERO 2007). Uma vez que um projeto de

pacotes ineficiente pode afetar negativamente a qualidade do sistema de software, um princípio proposto para guiar a organização de classes em pacotes, e base deste trabalho, é apresentado na subseção 2.3.5. Para se avaliar a qualidade de uma estrutura de pacotes, podem-se aplicar métricas de qualidade. Na seção 2.4, algumas dessas métricas são apresentadas. A seção 2.5 apresenta alguns trabalhos relacionados. A seção 2.6 apresenta as considerações finais deste capítulo.

2.2. A Estrutura "Pacote"

Software tende a evoluir ao longo do tempo, fazendo com que uma organização de alto nível facilite sua manutenção uma vez que os pacotes agrupam, idealmente, as classes correlatas, favorecendo ao entendimento do sistema através da navegação pelos seus arquivos fontes (GIBBS *et al.* 1990). Classes são muito granulares para serem usadas como a única unidade organizacional de grandes aplicações. Algo "maior" do que uma classe é necessário para organizar o volume de código fonte presente nestas aplicações. Na maioria das linguagens de programação modernas, esta estrutura é um classificador hierárquico de classes – que doravante passaremos a denominar "pacote".

Um pacote na Linguagem de Modelagem Unificada (UML) é usado "para agrupar elementos, e para fornecer um espaço de nomes para os elementos agrupados" (OMG 2007). Também em UML, um pacote pode conter outros pacotes, fornecendo assim a mesma organização hierárquica previamente mencionada. Porém, é importante notar que, na UML, um subsistema pode ser representado por um pacote, mas nem todo pacote é um subsistema. Pacotes permitem que classes sejam organizadas em abstrações nomeadas, mais genericamente referenciadas como subsistemas. Dentro de um sistema, podem existir subsistemas em diversos níveis de abstração (LAKOS 1996) e cada um deles pode ser particionado em novos subsistemas. Em resumo, pacote é uma estrutura que permite agrupar classes bem como outros pacotes de menor granularidade.

Linguagens de programação como Java, C++ e Ada suportam um alto-nível de organização através de estruturas para agrupamento de módulos de código (arquivos contendo o código fonte relativo a um programa específico). Tais estruturas podem ser implementadas de várias formas, dependendo da linguagem escolhida. Algumas linguagens possuem propriamente a estrutura de pacotes (*package*), enquanto outras a implementam através de arquivos ou mesmo estruturas de diretório. Java, por exemplo, permite tanto a declaração de mais de uma classe no mesmo arquivo de código como o agrupamento por diretórios. Linguagens do Framework Microsoft .NET,

como o Visual Basic e o C#, também permitem que várias classes sejam declaradas no mesmo arquivo de código (.vb ou .cs) – ou a definição de diversas interfaces. Todas estas implementações, entretanto, servem ao mesmo propósito: permitir organizar componentes básicos de programas (classes) segundo algum critério definido pelo programador. De uma forma genérica e unificada, pode-se utilizar o termo “pacote” usado pela OMG em sua linguagem de modelagem (UML) para especificar tais agrupamentos – como já foi definido no início desta seção.

Uma vez que o “pacote” é uma estrutura recursiva e hierárquica, é possível, ao analisarmos a organização dos mesmos, obtermos diferentes visões sobre o sistema, cada uma em um determinado nível hierárquico – tais visões podem estar relacionadas com um determinado nível de abstração do sistema em si. Tomemos como exemplo o construtor *package*, em Java. Este construtor é uma estrutura recursiva no sentido que pode conter classes e/ou outros pacotes. É a natureza recursiva do “pacote” que o permite representar subsistemas em níveis diferentes de abstração. Sendo assim, um dado pacote pode representar um subsistema em um nível mais alto de abstração do que os subsistemas representados pelos pacotes que ele contém.

Outra forma de visualizarmos os pacotes, considerando suas relações entre si, é através da análise das dependências implicitamente contidas neles. É comum que classes possuam dependências com outras classes. Estas dependências podem cruzar os limites do pacote, fazendo com que os pacotes tenham relacionamentos de dependência uns com os outros. Tais dependências, quando expressas graficamente, também permitem uma visão clara da hierarquia dos pacotes que compõe o sistema – como será examinado mais detalhadamente na subseção 2.3.4.

2.3. Projeto de Pacotes

A motivação para qualquer princípio de projeto – seja este relacionado a classes, interações entre objetos ou pacotes – é que a aplicação do princípio irá aumentar a qualidade do sistema de software de alguma forma. Com relação ao projeto de pacotes, alega-se que alocando classes em pacotes de acordo com os princípios descritos nas próximas subseções, o resultado será a construção de sistemas de maior qualidade (por exemplo, sistemas estruturados em componentes mais reutilizáveis, mais coesos e menos acoplados) do que se as classes tivessem sido alocadas de uma forma mais *ad-hoc*.

A disciplina de Projeto de Pacotes busca responder a perguntas como (MARTIN 2002): (a) Quais são os princípios para alocar classes em pacotes?; (b) Que

princípios de projeto governam os relacionamentos entre pacotes?; (c) Pacotes devem ser projetados antes das classes (*top down*) ou as classes devem ser projetadas antes dos pacotes (*bottom up*)?; (d) Como os pacotes são fisicamente representados?; e (e) uma vez criados, que propósito possuem os pacotes?

Muitos autores têm identificado princípios de projeto de pacotes. Porém, estes mesmos autores têm referenciado esses princípios usando termos diferentes. Lakos, por exemplo, usa o termo "projeto físico" para se referir coletivamente aos seus princípios para projeto de pacotes (LAKOS 1996). Martin usa o termo projeto de pacotes (MARTIN 2002) para agrupar uma série de princípios de organização de classes em pacotes, visando à produção de software de maior qualidade. Trabalhos anteriores em projeto de pacotes não costumam nomeá-lo explicitamente, mas usam termos como categoria de classes (BOOCH 1991), *clusters* (MEYER 1995), *subject areas* (COAD e YOURDON 1991), domínios (SHLAER e MELLOR 1992) e subsistemas (BOOCH 1987) para se referir as suas unidades fundamentais.

É importante notar que, embora projeto de pacotes não se relacione diretamente com a qualidade do código escrito, esta disciplina se propõe a facilitar, dentre outras tarefas, o teste e a manutenção de um sistema de software como um todo.

Em última análise, projeto de pacotes pode ser entendido como a atividade de planejar a organização de classes em subsistemas. Desta forma, linguagens de programação poderiam permitir o nível de abstração fornecido por subsistemas mesmo sem uma estrutura explícita de pacotes, implementando os subsistemas através de diretórios (sistema de arquivos) separados, que seriam os nomes que identificariam os subsistemas. Alternativamente, subsistemas podem ser compostos através de múltiplas declarações de classes em um único arquivo fonte (LAKOS 1996).

2.3.1. Mantendo Todas as Entidades Pequenas

Uma boa prática para construção de software orientado a objetos é manter as entidades (classes e pacotes) com um tamanho reduzido (BAY 2008). Esta recomendação pode ser observada tanto do ponto de vista da classe como de um ponto de vista de maior abstração, ou seja, considerando os pacotes.

Segundo tal recomendação, nenhuma classe deveria ter mais do que cinqüenta linhas e nenhum pacote deveria conter mais do que dez arquivos (BAY 2008). Classes com mais de cinqüenta linhas comumente fazem mais do que uma atividade específica, o que dificulta seu entendimento e reuso. Classes com até cinqüenta linhas também têm o benefício de serem visíveis em uma única tela, sem a necessidade de

rolagem, o que as torna fáceis de ser lidas. Classes ou pacotes, quanto menores, mais fácil sua assimilação.

Note-se que, isoladamente analisado, o tamanho (LOC) ideal de um módulo ou classe pode ser considerado um ponto controverso. Considerando, por exemplo, um sistema de software composto de 500.000 linhas de código, é possível uma configuração de 10.000 classes de 50 linhas ou 500 classes de 1000 linhas.

Uma abordagem para analisar tal questão é o princípio da coesão – assumindo que quanto mais coeso, mais compreensível um código fonte. Porém, há quem argumente que o tamanho de uma classe por si só não é um indicativo de coesão (COUNSELL *et al.* 2005). Ao contrário, o tamanho da classe (quando expresso em termos de LOC, ou mesmo número de métodos) não tende a influenciar a percepção de coesão. Se por um lado o conceito de coesão de software é bem conhecido e documentado, tanto no paradigma procedural quanto no orientado a objetos, por outro, engenheiros de software não possuem uma percepção clara do que empiricamente constitui uma "unidade" coesa (COUNSELL *et al.* 2005).

Além do fato de que elementos conceituais do domínio (classes) nem sempre são tão pequenos, o grande desafio em se criar classes menores é que geralmente existem grupos de comportamento que fazem sentido estar juntos. Este é o momento onde os pacotes se fazem úteis. À medida que as classes se tornam menores e possuem menos responsabilidades, e à medida que o tamanho do pacote é limitado, começa a ser visível que pacotes representam *clusters* de classes relacionadas que trabalham juntas para alcançar um objetivo (BAY 2008). Pacotes, assim como classes, devem ser coesos e ter um propósito bem definido. Manter estes pacotes pequenos os força a ter uma identidade real.

2.3.2. Tamanho Gerenciável

O limite para o tamanho do pacote deve ser definido (MELTON e TEMPERO 2007) e pode ser especificado em termos de número de classes diretamente ou indiretamente contidas em um pacote. O limite pode ser ditado pela política da empresa, preferência pessoal ou algum outro critério (ou métrica). Tais critérios se aplicam a qualquer limite no tamanho do pacote. O importante é que tal limite exista, ou seja, que o pacote tenha um tamanho gerenciável.

Também relacionado ao número ideal de classes por pacotes, é possível citar o "Número Mágico" de Miller (MILLER 1956). Em 1956, o estudo realizado por George Miller identificou que a quantidade de informação que pode ser lembrada após uma exposição é entre cinco e nove itens, dependendo da informação. Esta faixa é centrada de forma conveniente no número sete, o qual tem despertado o interesse de

pesquisadores há algum tempo. Aplicando uma faixa de mais dois ou menos dois, o número sete se torna conhecido como o Número de Miller (7±2). Considera-se que este é o número de itens que podem ser guardados na memória de curto prazo do ser humano médio adulto, por vez. *Short-term memory*, ou "memória de curto prazo", é um sistema para temporariamente armazenar e gerenciar informações necessárias para executar tarefas cognitivas complexas, como aprender, raciocinar e compreender (WEBSTER 1998). A "memória de curto prazo" está envolvida na seleção, inicialização e terminação de funções de processamento de informação, como codificação, armazenamento e recuperação de dados. No contexto de projeto de pacotes, este pode ser o critério utilizado para definição do tamanho de um pacote de classes gerenciável (COAD e YOURDON 1991).

O princípio conhecido como Tamanho Gerenciável (COAD e YOURDON 1991) está baseado no trabalho de Miller. Assim, pode ser argumentado que para um pacote ser rapidamente compreendido (usando a memória de curto prazo) ele deve conter de cinco a nove outros pacotes ou classes.

Outros autores divergem sobre este limite, embora identifiquem a necessidade de um limite para o tamanho do pacote. Lakos identifica 500 a 1.000 linhas de código (LOC) para um componente (subsistema de baixo nível) e de 5.000 a 50.000 LOC, ou algumas dúzias de componentes, por pacote (subsistema de alto nível) (LAKOS 1996). Meyer declara que um *cluster* deve conter de 5 a 40 classes e deve ser possível desenvolvê-lo com uma equipe de até 4 pessoas. Além disso, o cluster deve ser inteiramente compreensível por uma única pessoa (MEYER 1995).

2.3.3. Isolamento (*Stand-Alone*)

Um pacote deve ser tão independente quanto possível na medida em que ele deve ter dependência mínima de outros pacotes (LAKOS 1996). Um dado pacote depende de outro se suas classes não podem ser compiladas sem as classes do último.

A noção de dependência de compilação entre pacotes é importante porque é desejável que um pacote possa ser retirado de um programa para inserção em outro. Desta forma, é possível reusar código sem ter que modificar seu conteúdo textual para remover dependências. O isolamento de um pacote está diretamente ligado ao seu (baixo) acoplamento. Em Engenharia de Software, acoplamento (*coupling* ou dependência) é o grau em que cada módulo do programa precisa de cada um dos outros módulos (CONSTANTINE *et al.* 1974). É uma medida da interconexão entre os módulos de uma estrutura de software (PRESSMAN 1982). Sistemas com baixo acoplamento (*loosely coupled systems*) são desejáveis quando o sistema está sujeito

a mudanças freqüentes, uma vez que sua manutenção se torna mais fácil quando não existem tantas dependências.

Acoplamento fraco não é um conceito recente. Ao contrário, há décadas considerações e técnicas já foram propostas (CONSTANTINE *et al.* 1974) visando reduzir a complexidade de programas ao dividi-los em módulos com funcionalidades bem definidas. Isto possibilita criar sistemas complexos a partir de módulos simples, independentes e reusáveis. A depuração e a modificação de programas e o gerenciamento de grandes projetos de programação podem ser simplificados. E, à medida que a biblioteca de módulos cresce, programas ainda mais sofisticados podem ser implementados usando cada vez menos código novo.

Sendo assim, a propriedade de isolamento de um pacote também é importante para o entendimento, facilidade de teste e extensão do mesmo, na qual esforços de desenvolvimento em paralelo podem ocorrer através de diferentes equipes assumindo diferentes pacotes em um sistema (LAKOS 1996).

2.3.4. Propriedades Gráficas

O princípio de que um pacote deve ser independente leva a outros princípios de projeto de pacotes quando aplicado a todos os pacotes em um sistema. Estes princípios são auxiliares no sentido que certas características do que normalmente é referenciado como Grafos de Dependência de Pacotes (GDPs) implicam que os pacotes de um sistema não são independentes.

Um GDP é um grafo direcionado representando todos os pacotes no código fonte de um sistema em um dado nível de abstração como vértices e dependências entre estes pacotes como arestas direcionadas. Se o GDP de um sistema contém ciclos, ou é mais “alto” que “largo”, então os pacotes que compõem o sistema podem não ser tão independentes quanto se o grafo do mesmo fosse acíclico e “largo” (MELTON e TEMPERO 2007).

Como já foi dito na seção 2.2, pacotes têm dependências de compilação uns com os outros, tanto diretamente quanto indiretamente através de seus pacotes de menor granularidade que eles contém. É dito que o pacote “A” depende do pacote “B” se qualquer classe diretamente ou indiretamente contida em “A” depende de qualquer classe direta ou indiretamente contida em “B”. Para uma classe *C*, a relação DEPENDS-ON(*C*) é o conjunto de classes que devem estar disponíveis para que seja possível compilar *C* (MELTON e TEMPERO 2007).

O princípio conhecido como ADP (*Acyclic-Dependencies Principle*) (MARTIN 2002) declara essencialmente que não devem ser permitidos ciclos no GPD. Uma situação, referenciada com a expressão “síndrome da manhã-seguinte” (MARTIN

2002), é derivada do problema que ocorre quando, em uma equipe de desenvolvedores, um deles realiza alterações no final do dia em um módulo cujos demais desenvolvedores dependam. Estes deverão, no dia seguinte, adaptar-se às mudanças realizadas.

Os grafos na Figura 2.1 são GDP. É possível compará-los posto que eles possuem o mesmo número de subsistemas (vértices) e o mesmo número de dependências (arestas) – exceto por (a), que tem uma aresta extra. O propósito destes GDP é ilustrar que grafos cíclicos e altos tornam difícil comportar pacotes que são independentes. Logo, GDP devem ser largos e acíclicos. Se o grafo é cíclico, então todo pacote dependerá diretamente de seu "vizinho" e, indiretamente, dos demais pacotes do sistema – nenhum pacote conseguirá ser independente. Já se o GDP é "alto", existe uma tendência a que o único pacote independente seja o pacote da "base" do grafo – e quanto mais alto, maior a quantidade de pacotes de que se depende, culminando com o pacote do "topo", que depende dos demais pacotes do sistema.

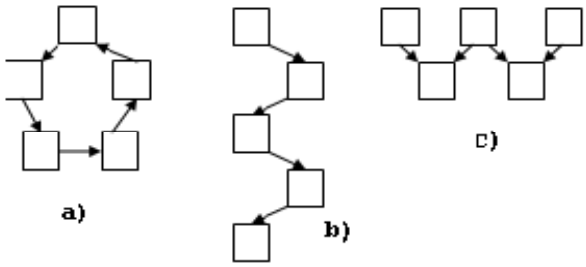

Figura 2.1. GDP cíclico, alto e largo

Considerando, inicialmente, qualquer pacote do GDP cíclico (a), para realizar uma extração para reutilização deste pacote em outro programa, também terão que ser copiados junto com ele todos os pacotes no grafo. Mesmo que o pacote só dependa diretamente de outro pacote, este outro pacote também depende de um terceiro pacote para ser compilado. Considerando o grafo "alto" (b), o pacote mais alto requer todos os outros pacotes para que possa ser entregue a outro sistema. O grafo "alto" (b) é melhor que o grafo cíclico (a) porque pacotes próximos da base do grafo transitivamente dependem de cada vez menos outros pacotes. O grafo "largo" (c) é melhor que o "alto" e o cíclico porque ele possui o maior número de pacotes que podem ser entregue com o número mínimo de outros pacotes. Sendo assim, cada pacote é mais "independente".

Um problema com os GDP da Figura 2.1 é que eles não são indicativos de projetos reais, porque projetos reais tendem a ter mais dependências diretas entre pacotes e tendem a ter mais "camadas" (MELTON e TEMPERO 2007). Lakos afirma

que um GDP que forme uma árvore binária balanceada (ver Figura 2.2) é um bom ponto de referência para se comparar projetos reais (LAKOS 1996), embora ele mencione que projetos reais não são tão regulares. Em termos de reuso, “folhas” da árvore são os pacotes mais independentes, uma vez que eles não dependem de pacote algum. Um quarto dos pacotes em tal árvore pode ser entregue com apenas dois outros pacotes, ou seja, cada pacote depende diretamente de apenas outros dois pacotes.

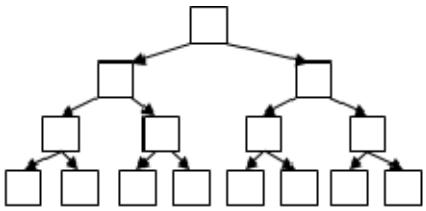

Figura 2.2. GDP em árvore binária balanceada

O debate sobre princípios de projeto para GDP, com base no acoplamento entre os módulos, tem justificado estes princípios à medida que pacotes no grafo devem ser independentes para que possam ser levados para outros sistemas (MELTON e TEMPERO 2007).

2.3.5. Princípio *Common-Closure*

Segundo Martin (2002), as classes em um pacote devem ser fechadas contra os mesmos tipos de modificações. Assim, uma mudança que afeta um pacote deve afetar todas as classes naquele pacote e nenhum outro pacote. Este princípio é chamado de *Common-Closure Principle*, ou PCC.

Na maior parte das aplicações, manutenibilidade é mais importante que reuso (MARTIN 2002). Se o código em uma aplicação deve ser modificado, é preferível que essa mudança ocorra em classes pertencentes a um único pacote, ao invés de ser distribuída por muitos pacotes. Se as mudanças são focadas em um único pacote, somente o pacote modificado precisará sofrer um novo *release* (distribuição). Outros pacotes, que não dependam do pacote modificado, não precisam ser revalidados, nem redistribuídos.

O PCC sugere que todas as classes que tendem a ser modificadas pela mesma razão sejam colocadas juntas em um mesmo pacote. Se duas classes são tão fortemente ligadas, quer seja fisicamente ou conceitualmente, de forma que elas sempre mudem conjuntamente, então elas devem permanecer juntas. Isto minimiza a sobrecarga de trabalho relacionada à geração de novos *releases*, revalidação e redistribuição do software.

2.3.5.1. Princípio da Responsabilidade Única

O Princípio *Common-Closure* (PCC) pode ser entendido como o Princípio da Responsabilidade Única (PRU) (MARTIN 2002) aplicado a pacotes – o PRU foi originalmente definido para classes (DEMARCO 1979). Assim como o PRU estabelece que uma classe não deve ter múltiplas razões para mudar, o princípio PCC diz que um pacote não deve ter múltiplas razões para ser modificado.

Segundo o PRU, uma classe deve possuir apenas uma razão para ser modificada. Quando os requisitos mudam, essa alteração irá se manifestar através de uma mudança na distribuição de responsabilidades entre as classes. Se uma classe assume mais de uma responsabilidade, então existirá mais de uma razão para que ela mude (MARTIN 2002).

Se uma classe tem mais de uma responsabilidade, então as responsabilidades se tornam acopladas. Mudanças em uma responsabilidade podem impossibilitar ou inibir a habilidade da classe contemplar as outras. Este tipo de acoplamento leva a projetos frágeis que “quebram” de formas inesperadas quando modificados.

No contexto do PRU, responsabilidade pode ser definida como “um motivo que pode gerar uma alteração em uma classe”. Se for possível pensar em mais de um motivo para se modificar uma classe, então aquela classe provavelmente tem mais de uma responsabilidade.

Dependendo de como a aplicação é modificada, as responsabilidades devem ser separadas ou não. Se a aplicação não é alterada de forma que as responsabilidades mudem em momentos diferentes, então não existe necessidade de separá-las. Em verdade, separá-las adicionaria uma complexidade desnecessária ao sistema. O corolário é: “Um eixo de mudança é um eixo de mudança apenas se a mudança realmente ocorrer” (MARTIN 2002). Não é prudente aplicar o PRU, ou qualquer outro princípio na verdade, se não existe sintoma.

2.3.5.2. Princípio *Open-Closed*

Ao projetar um sistema, é necessário que tal projeto seja estável na presença de mudanças e que vislumbre uma longa existência do sistema em si. Supondo que a evolução do sistema de software seja algo inevitável, é salutar que o projeto já leve em consideração tal expectativa, em especial quanto à sua possível extensão.

Segundo o Princípio *Open-Closed* (POC) (MEYER 1997), entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, porém fechadas para modificações. Quando uma única modificação em um programa resulta em uma cascata de mudanças nos módulos dependentes, o projeto possui um indício de rigidez. O POC adverte que tal sistema deve ser refatorado de tal forma que futuras

mudanças deste tipo não causem mais modificações em cascata. Se o POC é bem aplicado, então modificações são aplicadas adicionando novo código, não mudando código antigo que já funciona.

Ser aberto para extensão significa que o comportamento do módulo pode ser estendido. À medida que os requisitos da aplicação mudam, deve ser possível estender o módulo com novos comportamentos que satisfazem estas mudanças. Em outras palavras, é possível mudar o que o módulo faz.

Ser fechado para modificação significa que estender o comportamento de um módulo não resulta em mudanças no código fonte ou binário do módulo. A versão executável binária do módulo, quer seja uma biblioteca de vínculo dinâmico, uma DLL, ou um .jar Java, permanece intocada.

O PCC está fortemente associado com o POC, uma vez que o PCC é "fechado" no sentido definido pelo POC – classes "abertas" a um mesmo tipo de mudança contidas no mesmo pacote. Como já descrito anteriormente, o POC estabelece como as classes devem ser fechadas (para modificações), mas ao mesmo tempo abertas (para extensões) – já que alterações são inevitáveis, uma vez que "todos os sistemas mudam durante seu ciclo de vida" (JACOBSON *et al.* 1992).

Obter fechamento total não é uma tarefa fácil. Assim, o fechamento precisa ser estratégico. O projeto do sistema deve levar em consideração os prováveis tipos de mudança que devem ocorrer ou já foram observadas pela equipe de desenvolvimento. O PCC amplifica isto ao agrupar as classes que são abertas para certos tipos de mudanças nos mesmos pacotes. Assim, quando uma mudança nos requisitos é necessária, esta mudança tem uma boa chance de ser limitada a um número mínimo de pacotes – idealmente apenas um.

2.4. Métricas

Em se tratando de organização de classes, é útil determinar como os pacotes devem ser inter-relacionados. Uma forma de se gerenciar estas dependências entre pacotes é através de métricas. Tais métricas permitem aos desenvolvedores medir e caracterizar a estrutura de dependência de seus projetos.

Como foi exposto no capítulo 1, no contexto deste trabalho será feita uma análise a partir de métricas existentes relacionadas à organização e qualidade do software. O objetivo é avaliar como a reorganização das classes e pacotes (resultante da proposta objeto deste estudo) impacta na qualidade do software em si. As métricas revistas neste estudo se relacionam não apenas com as características do pacote em

si, mas utilizam também o viés da componentização – considerando pacotes como componentes do sistema de software.

2.4.1. Instabilidade

Esta métrica é derivada do Princípio das Dependências Acíclicas, descrito na subseção 2.3.4. A estabilidade de um pacote está diretamente relacionada com sua capacidade de acomodar modificações. Se modificar um determinado pacote não requer muito esforço, então este pacote tende a ser mais volátil. Por outro lado, um pacote difícil de modificar tende a ser mais estável, já que exigirá mais esforço para implementar supostas modificações (MARTIN 2002). Uma forma de tornar um pacote de software difícil de modificar é fazer outros tantos depender dele. Um pacote com várias dependências para ele se torna estável, posto que qualquer modificação nele irá exigir um trabalho extra de reconciliação com os demais pacotes que dependam dele.

Assim, uma forma de medir a estabilidade de um pacote é contar o número de dependências que entram e deixam aquele pacote – considerando-se o grafo de dependências de pacotes (MARTIN 2002). Esta contagem permite chegar ao cálculo da estabilidade posicional do pacote. As variáveis do cálculo em questão são representadas por: (A_e) acoplamento de entrada, ou seja, o número de classes fora de um pacote que dependem de classes dentro deste pacote; (A_s) acoplamento de saída, o número de classes dentro de um pacote que dependem de classes fora deste pacote; e (I) Instabilidade.

$$I = \frac{A_s}{A_e + A_s}$$

Figura 2.3. Métrica da instabilidade

Esta métrica retorna um valor no intervalo [0,1]. Nesta faixa, I = 0 indica um pacote estável ao máximo, ao passo que I = 1 indica a máxima instabilidade em um pacote. Por exemplo, um pacote que somente tenha outros pacotes dependendo dele – dito um pacote independente – terá um I = 0, posto que seu acoplamento de saída (A_s) será igual a zero. Logo, será considerado um pacote estável devido ao seu A_e.

O princípio da dependência estável (MARTIN 2002) nos diz que o índice I de um pacote deve ser maior que o índice I dos pacotes dos quais ele dependa. Se todos os pacotes de um determinado sistema forem totalmente estáveis, teremos um sistema incapaz de absorver mudanças. Posto isso, é desejável ter pacotes estáveis,

mas também possuir outros instáveis. A configuração ideal para pacotes é caracterizada por pacotes instáveis ($I \rightarrow 1$) dependendo de pacotes estáveis ($I \rightarrow 0$). Sendo assim, a medida de instabilidade deve diminuir na direção da dependência.

2.4.2. Abstratividade

Analisando os pacotes para classificá-los quanto à sua estabilidade, chegamos aos pacotes mais estáveis que, idealmente, compõem o núcleo do sistema – assumindo-se que o núcleo do sistema é a parte que menos tende a mudar ao longo do tempo. Tais pacotes, que são a base da arquitetura de um sistema, devem ser estáveis, posto que descrevem como o sistema foi projetado, ou seja, suas funcionalidades principais. Por outro lado, estes pacotes devem ser flexíveis o suficiente para não engessarem o software – o próprio projeto precisa de um mínimo de flexibilidade. A forma para se juntar dois conceitos – estabilidade e flexibilidade – que neste ponto poderiam ser vistos como opostos, é a utilização de classes abstratas: classes que são flexíveis o suficiente para serem estendidas sem a necessidade de modificações.

O Princípio das Abstrações Estáveis (MARTIN 2002) define a relação entre estabilidade e abstratividade. Para medir a abstratividade (A) de um pacote, usamos uma métrica que é calculada da seguinte forma: o (N_a) número de classes abstratas em um pacote dividido pelo (N_c) número total de classes naquele pacote.

$$A = \frac{N_a}{N_c}$$

Figura 2.4. Métrica da abstratividade

Esta métrica tem o limite [0,1]. Nesta faixa, $A = 0$ indica um pacote que não possui classes abstratas, ao passo que $A = 1$ indica a máxima abstratividade em um pacote, ou seja, ele só possui classes abstratas.

Uma vez estabelecida a relação entre a estabilidade de um pacote e sua abstratividade, podemos criar um gráfico com A no eixo vertical e I no horizontal. Se plotarmos dois tipos distintos de pacotes neste gráfico, verificaremos que pacotes que são totalmente estáveis e abstratos ficam no topo à esquerda em (0,1). Já os pacotes que são totalmente instáveis e concretos ficam na base à direita em (1,0), como pode ser visto na Figura 2.5.

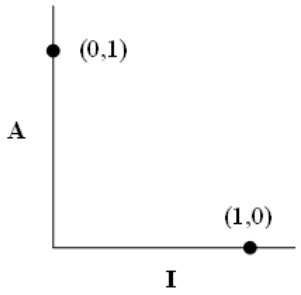

Figura 2.5. Pacotes estáveis e abstratos versus pacotes instáveis e concretos

Nem todos os pacotes se enquadram nestas duas posições, sendo o mais comum que os pacotes tenham níveis de abstratividade e instabilidade. Posto isso, podemos, usando este mesmo gráfico, assumir que existam posições aceitáveis para pacotes, bem como inferir áreas onde os pacotes não devam ficar (ou seja, zonas de exclusão). A Figura 2.6, a seguir, mostra as zonas de exclusão supracitadas.

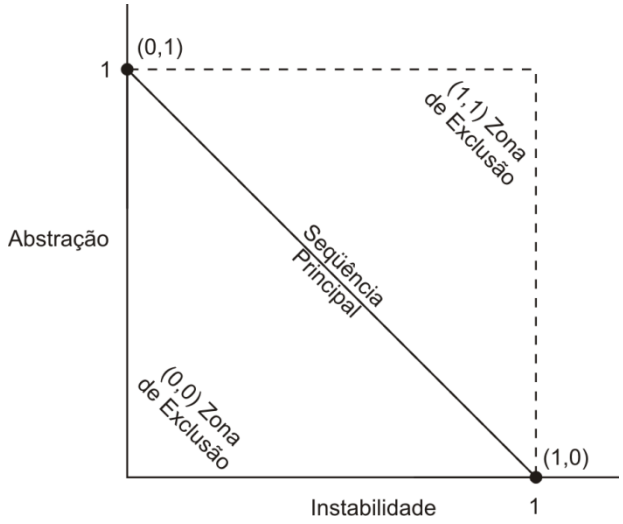

Figura 2.6. Zonas de exclusão

Considerando um pacote na área de (0,0), este é um pacote altamente estável e concreto. Tal pacote não é desejável porque ele é rígido. Ele não pode ser estendido porque ele não é abstrato. Adicionalmente, é muito difícil modificá-lo por causa de sua estabilidade. Assim, não é comum que pacotes bem projetados se encontrem próximo de (0,0). A área próxima de (0,0) é a zona de exclusão chamada “Zona da Dor” (MARTIN 2002).

Considerando um pacote próximo de (1,1), esta localização é indesejável porque ela é maximamente abstrata e ainda assim não possui dependentes. Tais pacotes são considerados inúteis uma vez que suas classes não possuem (em outros

pacotes) implementação. Assim, esta é chamada a “Zona da Inutilidade” (MARTIN 2002), já que não é possível instanciar objetos a partir de tais classes.

Torna-se claro que é desejável que os pacotes voláteis se encontrem o mais distante possível de ambas as zonas de exclusão, tanto quanto seja possível. O conjunto de pontos que são mais afastados de cada zona é a linha que conecta (1,0) e (0,1). Esta linha é conhecida como “seqüência principal” (ou *main sequence*) (MARTIN 2002). Um pacote que esteja na seqüência principal não é muito abstrato para sua estabilidade, nem muito instável para sua abstratividade. Não é nem inútil, nem particularmente “doloroso” de ser modificado. Ele é dependido na extensão de sua abstratividade e é dependente de outros na extensão de que é concreto.

2.4.3. Pacotes e Componentes

O Desenvolvimento Baseado em Componentes (DBC) se diferencia das outras abordagens de desenvolvimento de software através da separação entre a especificação do componente e a sua implementação e na divisão dos serviços dos componentes em interfaces (CHEESMAN e DANIELS 2001). As interfaces podem ser compreendidas como uma representação explícita das funcionalidades de um componente, sendo o meio de comunicação entre os mesmos. Deste modo, evita-se que mudanças na implementação de um componente afetem os demais, uma vez que os serviços continuam sendo garantidos pelas interfaces. A transformação de um software orientado a objetos para componentes pode torná-lo mais manutenível e reutilizável.

É sabido que o DBC tem um papel cada vez mais importante na indústria de software (BASS *et al.* 2001, WILLIAMS 2000). Isto ocorre devido à pressão econômica causada pela idéia de que DBC permite a redução de custo e do *time to market*, enquanto aumenta a qualidade do software através do reuso (SZYPERSKI *et al.* 2002). O raciocínio por trás disso é que a redução do custo pode ser obtida através da economia de escala, enquanto a melhoria na qualidade resulta do reuso de tais componentes em diferentes ambientes e aplicações.

Componentes de software são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e possuem interfaces claras, documentação apropriada e um grau de reutilização definido (SAMETINGER 1997). Considerando pacotes e componentes indistintamente, é possível aplicar as métricas relacionadas ao último para validar o *design* geral da aplicação no que tange a disciplina de projeto de pacotes.

Alguns trabalhos focam na análise do software a partir da coleta de métricas baseadas na organização geral de seus componentes. Por exemplo, foi realizado um

trabalho no sentido de propor *refactorings* a partir da análise de tais métricas (WERNER *et al.* 2008). As métricas utilizadas no trabalho, e que se aplicam a mensurar a qualidade geral de um software com relação à sua estrutura, foram:

(a) Número de Filhos por Superclasse para cada Componente (NOCC – *Number Of Children per Component Superclass*) adaptada de (CHIDAMBER e KEMERER 1994). Esta métrica possui um nível de granularidade de classe. Superclasses não devem possuir subclasses em um componente distinto do qual elas pertencem, pois este relacionamento faz com que um componente conheça detalhes da implementação do outro (VITHARANA *et al.* 2004). Esta métrica mensura o número de filhos da superclasse em cada componente. Sua fórmula pode ser descrita por $NOCC_k = \sum NOC_{ki}$, onde NOC_{ki} é o número de subclasses da superclasse k no componente i, onde k varia de 1 ao número total de superclasses do modelo e i varia de 1 ao número total de componentes do sistema. À luz da métrica da Abstratividade, descrita na subseção 2.4.2, e considerando a natureza recursiva da estrutura de pacotes, significa dizer que as subclasses concretas deveriam estar no mesmo componente, mas não necessariamente no mesmo pacote. As subclasses concretas estariam em um sub-pacote – considerando-se, neste caso, o componente sendo composto de um ou mais pacotes;

(b) Número de Implementações de Interfaces por Componente (NOIC – *Number Of Implementation per Component*) (WERNER *et al.* 2008), adaptada de (CHIDAMBER e KEMERER 1994). Esta métrica possui um nível de granularidade de classe. Classes que implementam uma mesma interface estão inseridas em um contexto comum e fornecem os mesmos serviços, de modo que uni-las deixará a funcionalidade em um único componente. Com relação ao princípio da Abstratividade, citado na subseção 2.4.2, as classes em questão podem se encontrar em pacotes menos abstratos uma vez que implementam interfaces. Por outro lado, não necessariamente todo o pacote é concreto, uma vez que a métrica considera apenas implementações, não importando se a classe é abstrata ou não – ou seja, podemos ter classes que, apesar de implementarem determinada interface, ainda assim são abstratas. Esta métrica mensura o número de classes que implementam uma determinada interface em cada componente. Sua fórmula pode ser descrita por $NOIC_k = \sum NOIC_{ki}$, onde $NOIC_{ki}$ é o número de classes que implementam a interface k por componente i, onde k varia de 1 ao número total de interfaces e i varia de 1 ao número total de componentes do sistema;

(c) Acoplamento entre Classes por Componente (CBOC – *Coupling Between Object Classes per Component*) (WERNER *et al.* 2008), adaptada de (CHIDAMBER e KEMERER 1994). Esta métrica possui um nível de granularidade de classe. O acoplamento da classe com relação ao componente ao qual ela pertence deve ser igual ou maior do que o acoplamento com relação aos demais componentes. Esta métrica mensura o valor de CBO (*Coupling Between Object Classes*) de cada classe k para cada componente i. O CBO de uma classe é o número de outras classes às quais ela está acoplada. CBO é um fator determinante para projetos modulares e pode impedir o reuso de uma determinada classe. Quanto mais independente uma classe é, mais facilmente pode ser reutilizada em outra aplicação (CHIDAMBER e KEMERER 1994). O CBO é utilizado na fórmula do CBOC, que pode ser descrita por $CBOC_k = \sum CBO_{ki}$, onde CBO_{ki} é o CBO de cada classe k em relação a classes do componente i, onde i varia de 1 ao número total de componentes do modelo e k varia de 1 ao número total de classes;

(d) Componentes Conectados (ConnComp – *Connected Components*) (SDMETRICS 2007). Esta métrica possui um nível de granularidade de pacote. A existência de grupos de classes isoladas no componente pode significar que o componente possui mais de uma funcionalidade, resultando em perda de coesão. Esta métrica mensura o número de grupos de classes conectadas, isoladas dos demais grupos, em um componente. Ela analisa, dado um componente, quantos grupos de classes possuem conexão entre si, mas não possuem nenhuma ligação com as demais classes no mesmo componente. Sua fórmula pode ser descrita por $ConnComp_i = TGCC_i$, onde $TGCC_i$ é o total de grupos de classes conectadas do componente i, onde i varia de 1 ao número total de componentes;

(e) Acoplamento por Componente Requerido (CRC – *Coupling per Required Component*) (BLOIS 2006). Esta métrica possui um nível de granularidade de pacote. A métrica verifica quantos componentes são necessários ao utilizar cada componente, pois o número de componentes requeridos deve ser o menor possível. A comunicação intercomponente é muito custosa (VITHARANA *et al.* 2004). A métrica faz um somatório de todos os componentes que possuem classes das quais o componente avaliado depende. Ela é calculada pela fórmula $CRC_i = TRC_i$, onde TRC_i é o total de componentes requeridos pelo componente i, onde i varia de 1 ao número total de componentes. Esta métrica está fortemente relacionada com a métrica da Instabilidade descrita na subseção 2.4.1, à medida que ela também pode ser utilizada para mensurar o nível de estabilidade de um

determinado componente – já que quanto maior o acoplamento, ou seja, o número de pacotes dependentes, mais estável tende a ser o pacote/componente.

Este conjunto não esgota as métricas de qualidade disponíveis para validação de componentes, mas ilustra algumas que podem ser aplicadas à validação da organização de pacotes do sistema de software. Outros trabalhos (GOULÃO e ABREU 2004, GILL e GROVER 2003) igualmente focam na avaliação de componentes. As estratégias de tais trabalhos podem ser igualmente aplicadas à avaliação de pacotes, como previamente exposto.

Goulão e Abreu (2004) focam na avaliação baseada em métricas da qualidade de componentes de softwares. Neste trabalho, eles focam na composição de componentes ao invés de focar em sua construção. Eles argumentam que avaliações em componentes *black-box* devem depender somente em informações publicamente disponíveis. Na verdade, nem sempre o código-fonte do componente estará disponível. Ao se utilizar tais componentes a maior preocupação não é com a complexidade interna do mesmo, mas com a complexidade envolvida em reusá-lo.

Gill e Grover (2003) também buscaram identificar linhas-guia para a definição de métricas que pudessem ser utilizadas na avaliação de sistemas baseados em componentes. O trabalho também discute algumas limitações do uso de métricas convencionais para a avaliação de componentes. Ainda assim, algumas de suas métricas podem ser utilizadas na avaliação de pacotes como, por exemplo, a *Component Interface Complexity Metric* (CICM), que basicamente está relacionada à complexidade das interfaces. É importante notar, entretanto, que este trabalho se preocupa mais em caracterizar uma família de métricas do que especificá-las de fato.

2.5. Mineração de Repositórios de Software

A análise de dados históricos buscando informações para tomada de decisão é uma área que tem recebido bastante contribuição de pesquisadores. Repositórios de software, tais como os mantidos por sistemas de controle de versão, bancos de dados de problemas, grupos de discussão e listas de correio, têm sido analisados no sentido de aprimorar o processo de desenvolvimento e evolução de software. Eles têm sido utilizados para descobrir informações previamente desconhecidas e avaliar teorias e abordagens de engenharia de software (KIM *et al.* 2006). Esta área de pesquisa é comumente chamada de mineração em repositórios de software (ou MSR – *Mining Software Repositories*).

A mineração de repositórios de software explora uma ampla gama de tópicos, incluindo a análise de co-mudança (BEVAN e WHITEHEAD 2003, BEYER e NOACK 2005, ZIMMERMANN *et al.* 2005), análise de origem (GODFREY e ZOU 2005, KIM *et al.* 2005a), análise de mudança de assinatura (KIM *et al.* 2005b), análise e previsão de defeitos (GRAVES *et al.* 2000), investigação de clones de código (KIM *et al.* 2005c), degeneração de código (EICK et al. 2001), estimativa de *drivers* para esforço de mudança em software (GRAVES e MOCKUS 1998) e qualidade (MOCKUS *et al.* 2005), identificação das principais características do processo de desenvolvimento de código aberto (MOCKUS *et al.* 2000), *chunking*[2] de software a fim de facilitar as equipes de desenvolvimento distribuído (MOCKUS e WEISS 2001) e construção de ferramentas para identificar desenvolvedores especialistas (MOCKUS e HERBSLEB 2002).

Mesmo que estes tópicos de pesquisa variem, todas as análises envolvidas precisam primeiro extrair dados a partir de repositórios de software. Desenvolver ferramentas de extração requer um esforço não trivial, especialmente para novos pesquisadores nesta área. A ferramenta Kenyon foi recentemente desenvolvida para simplificar a extração a partir de arquivos de versão (BEVAN *et al.* 2005). Entretanto, essas ferramentas ainda exigem o conhecimento sobre os sistemas de controle de versão e são, portanto, difíceis de aprender.

Uma das técnicas mais frequentemente usadas para minerar arquivos de versionamento é a co-mudança (ZIMMERMANN *et al.* 2006). A idéia básica é que itens que foram modificados juntos estão relacionados uns aos outros. Estes itens podem ser de qualquer granularidade: no passado co-mudança foi aplicada a mudanças em módulos (GALL *et al.* 1998), arquivos (BEVAN e WHITEHEAD 2003), classes (GALL *et al.* 2003) e métodos (ZIMMERMANN *et al.* 2003). Todas estas abordagens pararam na granularidade de métodos: aplicá-las em itens de granularidade mais fina, como blocos ou linhas de código, parece ser inviável, uma vez que eles são difíceis de identificar através das versões.

A mineração de dados também pode se valer da análise sobre as mudanças que ocorrem sobre os artefatos de software mantidos em sistemas de controle de versão, de forma a ajudar a manter até mesmo a documentação de um projeto de software atualizada (DANTAS *et al.* 2005). O processo de desenvolvimento de software engloba muitas fases distintas, cada uma delas trabalhando em um nível de abstração específico. Esta estrutura multi-nível é importante para modelar e refinar o

[2] "Chunking" é um método que se baseia na "quebra" da informação em pedaços mais maleáveis, facilitando assim a sua memorização (exemplo: telefone ou CPF).

conhecimento sobre um domínio de aplicação, proporcionando controle sobre a complexidade de desenvolvimento de software. Neste cenário, a notação UML é uma forte candidata para representar os artefatos de análise e projeto (PAGE-JONES 1999). No entanto, engenheiros de software devem assegurar que todos os artefatos UML estão atualizados e consistentes ao longo os diferentes níveis de abstração, para evitar mal-entendidos.

Técnicas de rastreabilidade podem ser usadas para resolver este problema através da identificação de todos os elementos do modelo UML que deverão ser atualizados quando uma mudança for introduzida (CLELAND-HUANG e CHANG 2003). Rastreabilidade de artefatos de software é um fator importante nos níveis de análise e projeto que auxilia engenheiros de software a identificar, e evitar, as más decisões de projeto no início do processo de desenvolvimento de software, além de fornecer uma visão de alto nível das dependências do sistema (SETTIMI *et al.* 2004).

2.6. Considerações Finais

Acredita-se que o projeto de pacotes tem um importante efeito na facilidade de reusar e de testar um software, tanto quanto em outros atributos de qualidade. Os princípios previamente mencionados buscam, por exemplo, descobrir se a organização das classes em um sistema o impede de ter pacotes que sejam independentes e que possam ser facilmente reutilizados. Os princípios também avaliam se os pacotes do sistema possuem tamanhos gerenciáveis, que facilitem seu entendimento e manutenção, além da distribuição do trabalho pela equipe de desenvolvedores.

O uso de métricas de qualidade busca suportar a avaliação da qualidade do sistema de software, tornando esta tarefa menos subjetiva possível. Além das métricas da disciplina de projeto de pacotes, é possível aplicar métricas da abordagem de desenvolvimento por componentes, desde que se assuma que pacotes e componentes são termos equivalentes.

Os princípios de projeto de pacotes são direcionamentos que objetivam guiar o projeto arquitetural de um sistema de software. Tais princípios prometem aumentar a qualidade final do produto sob diferentes aspectos como entendimento, manutenibilidade, facilidade de testes, distribuição e reuso. Tais princípios não são mutuamente exclusivos, mas ao contrário, são complementares e podem ser ajustados de acordo com a natureza do software em questão.

Na prática, porém, nem sempre os princípios de projeto de pacotes são utilizados e nem sempre eles convergem para o mesmo objetivo. Por exemplo, ao invés do PCC, é comum a utilização de agrupamentos por estereótipos, ou seja,

classes agrupadas em um mesmo pacote em função de realizarem tarefas do mesmo tipo (por exemplo, classes DAO, classes de negócio e classes responsáveis pela interface com o usuário). Se forem considerados modelos arquiteturais como o MVC, isto se torna mais visível. Cabe ao arquiteto responsável pela aplicação escolher o conjunto de princípios que se adéquam à sua realidade.

CAPÍTULO 3
REORGANIZANDO AS CLASSES

3.1. Considerações Iniciais

Neste capítulo será apresentada *ChangeFinder*, uma técnica para reorganização das classes de um sistema de software orientado a objetos, de modo a aumentar a aderência do sistema ao Princípio *Common Closure* (PCC) – princípio de projeto de pacotes descrito no capítulo 2. Esta técnica sugere uma organização para as classes de forma que os pacotes que formam o sistema passem a agrupar classes que sofreram modificações nos mesmos intervalos de tempo.

Considerando sua aplicação em organizações desenvolvedoras de software que utilizem um sistema de controle de versão para gerir o versionamento de seus artefatos, este trabalho analisa o problema da organização interna do software com base no histórico de alterações realizadas por seus desenvolvedores. A técnica utiliza o registro de alterações (*log*) sofridas pelas classes do projeto, mantido pelo sistema de controle de versão, para identificar que classes foram alteradas em um mesmo intervalo de tempo, discretizando o tempo total de desenvolvimento em intervalos de igual tamanho. Conhecidas as classes que foram alteradas em conjunto, a técnica move iterativamente cada classe para o pacote que contenha mais classes que foram alteradas conjuntamente com ela. Ao final, tem-se uma nova organização de classes em pacotes, de acordo com o PCC.

O restante deste capítulo está organizado da seguinte forma: na seção 3.2 é feita uma breve introdução aos sistemas de controle de versão e explicada a motivação para sua utilização na técnica proposta. Na seção 3.3 é definido o modelo do problema de reorganização de classes, além do vetor característico utilizado na técnica. Na seção 3.4 é apresentada a metáfora do “Espaço das Classes”, artifício usado para calcular a distância entre cada par de classes que compõem um sistema. Finalmente, na seção 3.5 o processo de reorganização é definido, explicando-se o algoritmo k-Médias, além de considerar aspectos relativos aos limites aceitáveis para o

número de classes em cada pacote. Terminando este capítulo, a seção 3.6 contém as considerações finais a respeito da técnica.

3.2. Sistemas de Controle de Versão

Um sistema de controle de versão VCS (do inglês, *version control system*), ou ainda SCM (do inglês, *source code management*), no contexto da Ciência da Computação e da Engenharia de Software, é um software com a finalidade de gerenciar diferentes versões no desenvolvimento de um documento (MOLINARI 2007). Esses sistemas são comumente utilizados no desenvolvimento de software para controlar as diferentes versões – histórico de desenvolvimento – do código-fonte e da documentação do sistema, sendo uma das atividades da área de Gerência de Configuração.

A Gerência de Configuração (GC) surgiu nos anos 50, sendo motivada pela necessidade da indústria aeroespacial norte-americana de controlar modificações na documentação referente à produção de aviões de guerra e naves espaciais (LEON 2000, HASS 2003, ESTUBLIER *et al.* 2005). Nas décadas de 60 e 70, a GC passou a abranger artefatos de software, indo além dos artefatos de hardware já estabelecidos e desencadeando o surgimento da Gerência de Configuração de Software (GCS) (CHRISTENSEN e THAYER 2002).

Pressman (2004) afirma que a gerência de configuração de software é o "conjunto de atividades projetadas para controlar as mudanças, através da identificação dos produtos do trabalho que serão alterados, estabelecendo um relacionamento entre eles, definindo o mecanismo para o gerenciamento de diferentes versões destes produtos, controlando as mudanças impostas, auditando e relatando as mudanças realizadas". Murta (2006) complementa dizendo que "a GCS não se propõe a definir quando e como devem ser executadas as modificações nos artefatos de software, papel este reservado ao próprio processo de desenvolvimento de software. A sua atuação ocorre como processo auxiliar de controle e acompanhamento". Ainda segundo Murta (2006), "o subsistema de GCS que mais recebeu contribuições, tanto de pesquisa quanto comercialmente, é o de controle de versões". Neste contexto, um sistema de controle de versão é uma ferramenta que provê a infra-estrutura para a definição de que artefatos devem ser controlados, bem como para gestão sobre "quem" realizou modificações nos mesmos. O código-fonte é o artefato de interesse neste trabalho.

Os sistemas de controle de versão comumente utilizam uma arquitetura cliente-servidor. Nesta arquitetura, o servidor armazena as versões atuais de todos os artefatos componentes de um projeto e o histórico de todas as versões destes

artefatos que foram criadas pelos desenvolvedores. Já os clientes se conectam a esse servidor para obter uma cópia completa do projeto, trabalhar nessa cópia e submeter novas modificações. Sistemas de controle de versão mais recentes mesclam o papel do servidor e do cliente, permitindo que os repositórios de artefatos sejam distribuídos entre diferentes computadores, atuando ora como cliente, ora como servidor. Por exemplo, GIT[3] é um software livre para controle de versão distribuído. Seu objetivo é atender a requisitos como desenvolvimento distribuído, manipulação de grandes conjuntos de arquivos, operações de junção (*merge*) complexas e alto desempenho nestes cenários (GIT 2010).

A terminologia comum do SCV considera um projeto como um conjunto de arquivos relacionados, gerenciados pelo SCV como um módulo do repositório. O módulo consiste de uma hierarquia de diretórios contendo os arquivos do projeto (CVS 2010). Um servidor SCV pode gerenciar diversos módulos simultaneamente.

Uma cópia do módulo que tenha sido baixada (através da operação de *checkout* ou *get*) para um cliente é chamada cópia de trabalho. À medida que o cliente realiza seu trabalho, ele provoca alterações nos arquivos componentes do projeto. Estas alterações devem ser salvas no repositório do SCV para que fiquem disponíveis para outros clientes (por exemplo, desenvolvedores). O salvamento é realizado através da operação de *check-in* ou *commit*, que salva a versão anterior de cada arquivo alterado e gera uma nova revisão do mesmo[4]. Assim, o histórico mantido pelo SCV é uma seqüência de revisões para cada arquivo de um módulo armazenado no servidor. Por controlar as operações acima, os SCV são capazes de manter um *log* que registra todas as operações aplicadas sobre os arquivos componentes do sistema.

A técnica de reorganização das classes que formam um sistema apresentada neste trabalho se baseia em um princípio de projeto que leva em consideração a modificação concomitante das classes para distribuí-las em pacotes. Desta forma, um sistema de controle de versão é um ponto de partida ideal para a aplicação da técnica, atuando como origem dos dados necessários para a sua execução: o histórico de modificações das classes componentes do sistema. Assim, é possível saber, para cada classe, quando a mesma foi alterada e comparar as datas de alteração desta classe com as datas de alteração das demais classes do sistema.

[3] http://git-scm.com/

[4] Algumas estratégias de versionamento não salvam o arquivo inteiramente, mas apenas o "diferencial" para a nova versão.

Neste trabalho foi utilizado o sistema de controle de versão CVS, devido à quantidade de projetos de código aberto existentes no site SourceForge[5] baseados neste sistema. O CVS[6], ou *Concurrent Version System*, é um sistema de controle de versão que permite trabalhar com diversas versões de arquivos, organizados em diretórios (ou módulos) e localizados no próprio sistema de arquivos do usuário ou remotamente (em um repositório). Ele permite que sejam mantidas as versões antigas dos artefatos componentes de um sistema e gera *logs* com registros de quem (e quando) manipulou estes arquivos. O sistema é especialmente útil para se controlar versões de um software durante seu desenvolvimento e fornecer um histórico da evolução do mesmo.

Como projetos orientados a objetos são compostos de “classes” e sistemas de controle de versão gerenciam “arquivos”, é importante ressaltar a relação entre estes dois termos e suas implicações para a técnica proposta. Neste trabalho foi assumido que um arquivo de código-fonte contém uma única classe ou, caso o arquivo contenha mais de uma classe, todas as suas classes sofrem alterações em conjunto. Desta forma, a análise das modificações que afetaram o sistema (informação necessária para aplicar a técnica proposta) será realizada sobre os seus arquivos e não diretamente sobre as suas classes. Doravante, classes e arquivos serão tratados indistintamente e consideraremos que as datas de alteração dos arquivos são as datas de alteração das classes.

Para efeito da técnica de reorganização de classes, os termos “pacote” e “diretório” também serão considerados equivalentes. Pacotes estão relacionados aos diretórios de tal forma que o primeiro é uma estrutura de organização lógica da aplicação, enquanto o segundo é uma estrutura de organização lógica do sistema de arquivos que mantém o código-fonte da aplicação. Algumas linguagens orientadas a objetos exigem uma relação mais estreita entre diretórios e pacotes. Em Java, por exemplo, cada diretório representa um pacote, ou seja, os arquivos que estão armazenados em determinado diretório representam artefatos (geralmente classes) que pertencem ao pacote associado ao diretório. Estas linguagens de programação reforçam a premissa de equivalência entre os dois termos. Por outro lado, mesmo linguagens que não impõem explicitamente esta exigência (C#, por exemplo) consideram boa prática manter todas as classes de um pacote a partir de um mesmo diretório, de forma a organizar melhor o código (MAYO 2000). Assim como classes e arquivos, doravante os termos “pacote” e “diretório” serão intercambiáveis.

[5] http://sourceforge.net/

[6] http://cvs.nongnu.org/

Com base nestas equivalências, os dados de entrada para a utilização da técnica proposta são os arquivos e as modificações que os afetaram, conforme registrado no *log* do sistema de controle de versão. Desta forma, apesar do CVS ter sido escolhido como ferramenta para obtenção dos dados de entrada utilizados nos estudos que foram realizados para avaliar a técnica proposta (ver próximo capítulo), outros sistemas de controle de versão, como o Subversion[7], podem ser utilizados – este último, inclusive, é considerado um substituto moderno ao CVS (MASON 2004, SANDLER 2006). Sendo assim, no escopo deste capitulo, será utilizado o termo Sistema de Controle de Versão (SCV) em referência à família de softwares que permitem a gestão de versões dos artefatos componentes de um sistema, e não obrigatoriamente ao CVS em si.

3.3. Modelo do Problema de Reorganização de Classes

Seja C o conjunto de classes compondo o projeto, com c ≥ 1 elementos. Cada classe é unicamente identificada por seu nome[8]. 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 ≥ 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 ≥ 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 realizou a alteração (a_k) e a data/hora quando a operação de *commit* que gerou esta revisão foi executada (d_k). Dessa forma, $r_k = [cr_k, id_k, a_k, d_k]$.

Para efeito de coleta de dados históricos sobre as modificações sofridas pelos arquivos componentes do sistema sob análise, o primeiro passo consiste em obter o *log* de alterações (até a presente data) a partir do SCV. Apesar de não ser necessário em um primeiro momento, o código-fonte do sistema pode ser obtido a partir de um *checkout* do módulo desejado no repositório do SCV.

A análise do *log* inicia com a identificação dos arquivos que compõem o projeto. Durante este processo de identificação, deve ser extraído do *log* o nome e o

[7] http://subversion.tigris.org/

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

"caminho" para cada arquivo no repositório do SCV. No contexto desse trabalho foram considerados somente os arquivos de código-fonte, ou seja, arquivos com extensões específicas das linguagens de programação utilizadas (.java, .cpp, .vb, .cs, dentre outras).

Após a identificação dos arquivos componentes do sistema, deve-se realizar a leitura das revisões de cada arquivo. Para cada revisão deve ser lido seu identificador, o nome do usuário que realizou a operação *commit* e a data/hora em que esta operação foi executada. Vale notar que a data da primeira revisão é a data em que o arquivo foi **registrado** no SCV. Outra informação útil, geralmente representada junto à revisão no *log* do SCV, é o "estado" do arquivo. O estado indica, por exemplo, se o arquivo foi removido do sistema. Portanto, o estado da última revisão de cada arquivo deve ser levado em consideração para excluir do conjunto de análise as classes que foram retiradas do sistema ("state: dead", no CVS, a título de exemplo).

Para a técnica caracterizar uma organização de classes em pacotes em linha com o proposto pelo PCC, é preciso criar uma medida de distância que compare duas classes em relação à ocorrência de mudanças sobre estas classes nos mesmos intervalos de tempo. Desta forma, é preciso descrever cada classe de acordo com seu perfil de mudanças, para posteriormente avaliar quão próximas duas classes estão entre si. Para representar uma classe será utilizado um vetor descritivo do seu padrão de modificação, chamado de vetor característico.

O vetor característico de uma classe representa todo o período de desenvolvimento desta classe, da data em que ela foi registrada no repositório do SCV até a data atual. Este período é dividido em intervalos iguais, consecutivos e não intercalados, cada qual representado por uma célula do vetor – o tamanho do intervalo (hora, dia, semana, mês, entre outros) é um parâmetro da técnica de reorganização de classes. Considerando o vetor como um conceito geométrico, representado em um espaço com determinado número de dimensões (R^2, R^3, ..., R^N), e que ele será utilizado como mecanismo de cálculo da distância entre as classes (próxima seção), o tamanho do intervalo é determinante para identificarmos o número de dimensões necessárias para representar o vetor característico.

Por exemplo, se o tempo de desenvolvimento de uma classe for dividido em três intervalos de tempo, seu vetor característico será representado em um espaço de três dimensões (R^3), posto que cada célula do vetor representa um intervalo de tempo. Em um segundo exemplo, considere uma classe que foi desenvolvida ao longo de 10 dias. Seu vetor característico, considerando um intervalo diário, terá 10 células e será representado em um espaço de 10 dimensões (R^{10}). A Figura 3.1 apresenta um exemplo de vetor característico para uma classe fictícia.

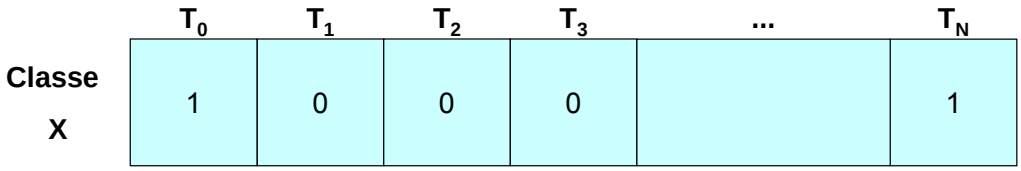

Figura 3.1. Exemplo de vetor característico para uma classe

Cada célula do vetor característico de uma classe representa um intervalo de tempo onde o arquivo que contém a classe pode ter sofrido alguma alteração. Se a classe foi alterada em um determinado intervalo, a célula correspondente em seu vetor característico possuirá valor **um**. Caso contrário, a célula possuirá valor **zero**. Uma vez obtido o *log* do sistema de controle de versões, pode-se processá-lo para a criação do vetor característico de cada classe.

Além do tamanho do intervalo de tempo, o processo de construção do vetor característico deve levar em consideração um limite de tamanho do *log*. Sistemas de software muito antigos podem possuir *logs* de tamanho considerável (chegando a megabytes ou referentes a uma década ou mais de *commits*). Neste caso, o arquiteto pode considerar que as alterações mais antigas não refletem o perfil de alteração que o sistema vem sofrendo recentemente. Assim, para que a organização de classes reflita o perfil atual de alterações simultâneas, o arquiteto pode executar a técnica limitando a análise do *log* apenas a revisões realizadas após uma determinada data.

Outro filtro possível seria um número mínimo de revisões para um arquivo ser considerado na análise. Arquivos que sofreram poucas revisões tendem a ser arquivos incorporados mais recentemente ao repositório ou que foram reutilizados e pouco alterados no contexto do projeto. Estes arquivos podem ter pouco a contribuir no sentido de identificar o perfil de mudanças do sistema. Dessa forma, o arquiteto pode optar por considerar em sua análise apenas os arquivos que sofreram um número mínimo de revisões ao longo do tempo de desenvolvimento.

Ao final do processamento do *log*, existirá um vetor característico para cada classe. O conjunto desses vetores formará uma matriz bidimensional de números inteiros, onde uma dimensão está relacionada aos intervalos de tempo onde as modificações podem ter ocorrido (de acordo com o parâmetro que determina o tamanho do intervalo de tempo) e a outra dimensão está relacionada aos arquivos de código-fonte que compõem o sistema. Cada célula da matriz indica se determinado arquivo sofreu ou não modificação em um intervalo de tempo. A Figura 3.2 apresenta um exemplo de matriz formada pelos vetores característicos das diversas classes que compõem um sistema.

Intervalo	Classe A	Classe B	Classe C	Classe X	...
T_1	1	0	1	1	...
T_2	1	1	1	0	...
T_3	1	0	1	0	...
T_4	1	0	0	0	...
T_5	1	1	0	0	...
T_6	1	1	0	0	...
...	...	...	...	...	...
T_N	0	0	0	1	...

Figura 3.2. Exemplo de matriz formada pelos VCs das classes do sistema

A técnica de reorganização de classes em pacotes parte de um conjunto P_{ANTES} inicial, construído a partir da organização atual do projeto, e propõe um conjunto P_{DEPOIS}, com os mesmos arquivos do conjunto inicial, porém distribuídos entre pacotes de modo a aumentar a aderência do projeto ao PCC. Com base nestes parâmetros e dados de entrada, a técnica de reorganização de classes pode ser formalmente descrita como:

$$f: (C, P_{ANTES}, R, \Delta, d_{INICIO}, r_{MIN}) \rightarrow P_{DEPOIS}$$

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 a partir do *log* do sistema de controle de versão com o histórico de alterações;

Δ é o tamanho do intervalo de tempo em que serão analisadas as alterações conjuntas (usado para construir os vetores característicos);

d_{INICIO} é a data inicial para se considerar as revisões $r \in R$;

r_{MIN}, é o número mínimo de revisões para que o arquivo seja considerado no processo de análise; e

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

3.4. Uma Medida de Distância entre as Classes

Uma vez conhecidos os vetores característicos das classes, o próximo passo é estabelecer um mecanismo que permita a reorganização do conjunto de arquivos de acordo com a freqüência com que estes arquivos foram modificados nos mesmos intervalos de tempo. Para tanto, como já foi dito, é necessária uma medida de distância que considere que duas classes estão próximas quando a maior parte de suas mudanças ocorre nos mesmos intervalos. Por outro lado, se o perfil de mudança das classes indica que elas sofrem alterações em intervalos distintos, as classes devem ser consideradas distantes.

Considerando que cada classe é representada por um vetor característico, o "Espaço das Classes" é a metáfora utilizada para ilustrar o espaço geométrico onde estes vetores podem ser representados e onde pode ser calculada a medida de distância entre classes.

O "Espaço das Classes" é um espaço euclidiano ($\mathbb{R}^n$) onde cada dimensão está relacionada a um intervalo de tempo. O valor da célula de cada vetor característico em um intervalo indica o valor da coordenada deste vetor na dimensão referente a este intervalo. Como o valor da célula pode ser somente zero ou um, todos os vetores característicos estão representados nos vértices de um hiper-cubo de lado igual a um, situado dentro do primeiro quadrante do "Espaço de Classes". A Figura 3.3 apresenta este hiper-cubo para um espaço de tridimensional.

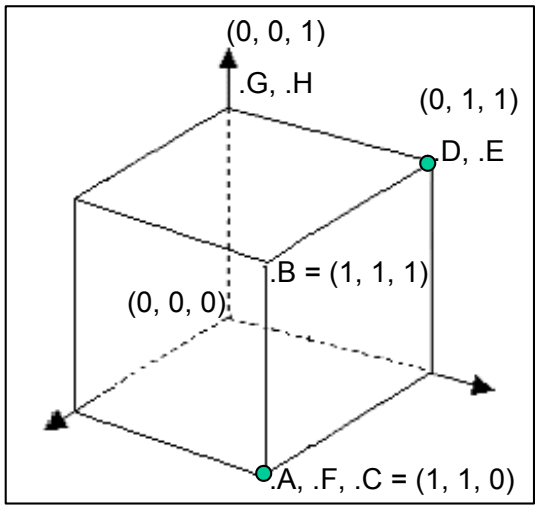

Figura 3.3. Exemplo de espaço das classes em R^3

Por exemplo, na Figura 3.3 considere os arquivos "D" e "E" que **não** sofreram modificações no primeiro intervalo de tempo, mas foram alterados no segundo e terceiro intervalos. De acordo com a regra de representação dos seus vetores característicos, ambos devem se localizar no mesmo ponto do espaço tridimensional

(três intervalos de tempo). Por outro lado, considerando os arquivos “D” e “A” (o arquivo “A” sofreu alterações apenas no primeiro e segundo intervalos de tempo), vemos que os arquivos estão em vértices opostos de uma face do hiper-cubo.

Considerando a representação das classes neste espaço, a distância entre duas classes será a distância euclidiana (BOLDRINI *et al.* 1980) entre os vetores característicos que as representam. A distância euclidiana (ou distância métrica) é a distância entre dois pontos $P = (p_1, p_2, ..., p_n)$ e $Q = (q_1, q_2, ..., q_n)$ em um espaço euclidiano n-dimensional, definida pela equação a seguir:

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

A exemplo de outros trabalhos (MONETA *et al.* 1990, ASADA *et al.* 1992, XAVIER *et al.* 2002, DIAS-NETO 2009), a técnica proposta utiliza uma abordagem vetorial para a avaliação de distâncias conceituais. Por exemplo, XAVIER *et al.* (2002) utiliza a distância euclidiana para apoiar a seleção de padrões arquiteturais em projetos de software, explorando a distância entre o padrão desejado por um projeto e os padrões arquiteturais disponíveis a partir da comparação de representações destes elementos em um espaço vetorial multidimensional. Considerando novamente a Figura 3.3, é possível verificar que a distância entre os arquivos “D” e “E” é igual a zero, enquanto a distância entre os arquivos “D” e “A” é igual a $\sqrt{2}$.

Ao calcular a distância euclidiana para cada par de classes, é possível construir uma matriz de distâncias, onde cada linha, bem como cada coluna, representa uma classe e o valor de cada célula contém a distância entre as classes representadas na linha e coluna da célula. Essa matriz é simétrica e possui diagonal zero (pois a distância entre uma classe e ela mesma é sempre zero), como pode ser visto na Figura 3.4 que utilizou o cenário fictício da Figura 3.3.

	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 3.4. Exemplo de matriz de distâncias

Considerando que todas as classes estão representadas neste espaço multidimensional a partir de seus vetores característicos, a técnica proposta busca

representar os pacotes neste mesmo espaço de maneira que seja possível calcular a posição de um pacote de acordo com a posição das classes contidas nele. Igualmente, a distância de uma classe a um pacote é a distância euclidiana entre o vetor característico da classe e o vetor característico do pacote, que será identificado pela técnica.

3.5. Classificando as Classes em Pacotes

A identificação das classes que fazem parte de cada pacote é um problema de agrupamento. O termo Análise de Agrupamentos, primeiramente usado por Tyron (1939), comporta uma variedade de algoritmos de classificação, todos voltados para uma questão importante em várias áreas da pesquisa: como organizar um conjunto de elementos em grupos, de forma que cada grupo represente um subconjunto de elementos similares entre si e distintos dos demais? Ou como desenvolver taxonomias capazes de classificar dados observados em diferentes categorias (WANGENHEIM 2006).

Biólogos, por exemplo, têm de organizar dados observados em grupos que tenham similaridades de acordo com determinados critérios, ou seja, desenvolver taxonomias. Zoologistas confrontados com uma variedade de espécies de um determinado tipo, por exemplo, devem primeiramente classificar os espécimes observados em grupos para que então seja possível descrever esses animais em detalhes, destacando as diferenças entre espécies e subespécies.

A atividade de agrupamento pode ser vista como um processo dirigido pelos dados observados, de forma a agrupar esses dados segundo características comuns que ocorram neles (WANGENHEIM 2006). Este processo deve levar em conta a possibilidade de se realizar, inclusive, uma organização hierárquica de grupos. Assim, a cada nível maior de abstração também são maiores as diferenças entre elementos contidos em cada grupo – da mesma forma que espécies de animais do mesmo gênero têm muito em comum entre si, mas espécies de animais que possuem apenas o filo ou a ordem em comum possuem pouca similaridade.

Existem diversas soluções para resolver problemas de agrupamento quando os elementos a serem agrupados estão representados como vetores ou pontos e os grupos em que estes elementos serão classificados estão descritos da mesma forma. Há dois algoritmos de agrupamento de dados baseados em métodos estatísticos interessantes para efeitos de classificação de padrões (WANGENHEIM 2006): “Unificação” (ou “Agrupamento em Árvore”) e “Agrupamento por k-Médias”. Destes, foi escolhido o algoritmo de k-Médias, que será explicado na próxima subseção.

3.5.1. O Algoritmo de k-Médias

O processo de reorganização utilizado na técnica proposta é uma variação de um algoritmo de *cluster analysis* conhecido como k-Médias (*k-Means*). *K-means* (MACQUEEN 1967) é um dos mais simples algoritmos de aprendizado não supervisionado que resolvem o problema de agrupamento. Em estatística e *machine learning*[9], k-Médias *clustering* é um método de *cluster analysis* que particiona *n* observações em *k clusters,* nos quais cada observação pertence ao *cluster* com a média (dos valores representativos dos elementos contidos naquele *cluster*) mais próxima (WANGENHEIM 2006). O agrupamento por k-Médias é um método de Análise de Agrupamentos **não-hierárquico por repartição**. Ele assume que já é conhecido o número de conjuntos em que os dados serão particionados (WANGENHEIM 2006).

O algoritmo básico do método das k-Médias pode ser descrito assim:

1. Padronizam-se todos os dados, descrevendo-se cada variável em termos da distância de seu valor em desvios-padrão da sua média;
2. Fixa-se o número de agrupamentos desejados = *k*;
3. Dividem-se os casos aleatoriamente nos *k* grupos;
4. Calcula-se o centróide[10] de cada grupo;
5. Calcula-se, para cada caso, a distância euclidiana em relação ao centróide de cada grupo;
6. Transfere-se o caso para o grupo cuja distância ao centróide é mínima;
7. Repete-se (4), (5) e (6) até que nenhum caso seja mais transferido.

É importante salientar que, embora possa ser provado que o processo irá sempre terminar, o algoritmo de k-Médias não necessariamente encontra a melhor configuração. Também vale ressaltar que, como foi exposto, o algoritmo de k-Médias depende de um conhecimento prévio do número de grupos. Como na técnica de reorganização de classes os grupos são os pacotes, é preciso definir previamente o número de pacotes a ser utilizado pela técnica. Este é o assunto tratado na próxima subseção.

[9] *Machine learning* é uma disciplina científica que se ocupa do projeto e desenvolvimento de algoritmos que podem mudar de comportamento em função de dados, tais como dados de sensores, ou de banco de dados (MITCHELL 1997).

[10] Ver subseção 3.5.3

3.5.2. Definindo o Número de Pacotes

Embora a literatura não seja clara em indicar o número ideal de classes por pacote, alguns trabalhos oferecem diretivas para obter um tamanho razoável nesta relação. Como foi exposto nas seções 2.3.1 e 2.3.2 do capítulo 2, é recomendável que as entidades (em específico, os pacotes) sejam mantidas pequenas de forma a obter um software mais facilmente gerenciável.

Na ausência de uma diretriz mais clara, optou-se por fazer um levantamento em softwares de código aberto, buscando indícios do que seriam tamanhos de pacotes utilizados "na prática". Após a análise de alguns softwares de código aberto, foram obtidos os resultados listados na Tabela 3.1.

Tabela 3.1. Levantamento sobre Tamanho Médio de Pacotes

Software	Classes Por Pacotes (Média)	Classes Por Pacotes (Mínimo)	Classes Por Pacotes (Máximo)
JDK6	16,79	1,00	216,00
JDK1.4	19,48	1,00	182,00
Eclipse	7,70	1,00	162,00
NetBeans	6,34	1,00	129,00
JBoss	6,16	1,00	78,00
Azureus	5,50	1,00	31,00
jVI	16,60	3,00	53,00
FMJ	4,55	1,00	88,00
OpenOffice	6,81	1,00	126,00
Média (exceto JDK6, JDK1.4 e jVI)	**6,17**	-	-
Média (Todos)	**9,99**	-	-

Ao contrário do que poderia se imaginar, a maior parte dos softwares analisados possuía diversos pacotes com apenas **uma** classe. Já o número médio de classes por pacote ficou em torno de 6, se forem desconsiderados a API do Java (JDK versões 4 e 6) e o editor jVI – ambos possuem um número reduzido de pacotes, fazendo com que a média de classes por pacote fique mais elevada (em torno de 16). Também foi verificado que, nestes softwares, é possível encontrar pacotes com um número elevado de classes, chegando até 216 no caso do JDK6. A existência de pacotes com muitas classes não é considerada boa prática, pois dificulta o gerenciamento do mesmo. Em última análise, se considerarmos todos os softwares do levantamento, a média é de quase 10 classes por pacote.

Os valores encontrados para a quantidade de classes por pacote, ou mesmo os valores ditos ideais segundo a literatura, podem ser adotados caso o arquiteto não esteja satisfeito com a quantidade de pacotes da estrutura atual do software. Neste trabalho, entretanto, optou-se por utilizar a estrutura atual de pacotes como ponto de

partida. Assim, o número de pacotes utilizados no processo de reorganização será o mesmo da distribuição atual do software – considerando todos os pacotes que possuam ao menos uma classe.

3.5.3. Aplicando o k-Médias sobre Classes e Pacotes

Uma diferença entre o algoritmo utilizado na técnica proposta e o algoritmo original de k-Médias é que, neste último, o passo inicial consiste em distribuir aleatoriamente os elementos pelo número desejado de conjuntos, enquanto na técnica proposta optou-se por utilizar a disposição original das classes em pacotes como ponto de partida. Isto foi feito com o objetivo de, após aplicar a técnica, oferecer sugestões de *refactoring* a partir do projeto original – partindo da organização de classes em pacotes que o arquiteto definiu, ao invés de se desconsiderar suas escolhas. Sendo assim, a primeira atividade da técnica proposta consiste em criar tantos conjuntos (pacotes) quanto o número original de pacotes não vazios e distribuir as classes de maneira que cada uma fique em seu pacote original.

É importante notar que ao contrário da estrutura hierarquizada da configuração original, a configuração de pacotes a ser submetida ao algoritmo é uma configuração “flat”, posto que o algoritmo k-Médias não contempla a criação de hierarquias. No escopo deste trabalho, não se levou em consideração o fato de um pacote estar contido em outro – todos os pacotes são dispostos em um mesmo nível. Sua organização de forma hierárquica será objeto de trabalhos futuros.

O processo de reorganização dos arquivos em seus pacotes ideais é iterativo e, sempre que uma nova disposição de arquivos for estabelecida, deve-se recalcular o posicionamento dos centróides. Um centróide é uma referência para o centro de um pacote, conforme o espaço em que ele se encontra representado. Calculados os centróides, é possível reiniciar a busca por um pacote mais próximo para cada classe. Neste contexto, cada centróide, assim como as classes, é representado por um vetor característico.

Para identificar se uma classe deve permanecer em seu pacote ou mudar de pacote, calcula-se a distância desta classe a todos os pacotes e identifica-se o pacote mais próximo a ela. As classes que possuem pacotes mais próximos diferentes do pacote onde se encontram são movidas conjuntamente ao final da iteração. Ao final de uma iteração por todas as classes, o centróide de cada pacote é recalculado como a média simples dos vetores característicos das classes que pertencem a este pacote. Se pelo menos uma classe trocar de pacote, a iteração irá se repetir. A Tabela 3.2 exemplifica um cenário onde uma determinada “classe X” está mais próxima do “pacote 1” do que do “pacote 2”.

Tabela 3.2. Cálculo da Distância Classe – Centróide

Intervalo	Classe "X"	Centróide(Pacote) "1"		Centróide(Pacote) "2"
1	1	0,8		0,5
2	0	0,2		0,4
3	1	1		0
4	0	0,3		1
5	0	0,5		0,2
6	1	0,8		0,8
7	1	0,9		0,2
8	0	0,2		0,5
9	0	0,5		0,5
10	0	1		0,5
Distância:		1,326649916		1,96977156

Como pode ser visto na Tabela 3.2, a classe fictícia “X” é representada por um vetor característico com 10 intervalos. Considerando dois pacotes “1” e “2”, cada qual com seu vetor característico, é possível calcular a distância da classe “X” para cada um dos pacotes através da distância euclidiana entre os vetores envolvidos. Por este cálculo, verifica-se que a distância de “X” para o pacote “1” (1,32) é menor do que a distância de “X” para o pacote “2” (1,96). Assim sendo, ao final da iteração, a classe “X” deverá ser movida para o pacote “1”. Quando não houver mudanças durante uma iteração, diz-se que o conjunto convergiu para uma disposição onde cada classe se encontra em seu pacote ideal segundo o PCC.

3.5.4. Considerando os Limites de Tamanho dos Pacotes

Embora o PCC em si não faça nenhuma menção ao tamanho do pacote, alguns autores (BAY 2008, LANZA e MARINESCU 2006) argumentam que o ideal é que o pacote tenha um tamanho reduzido ou que exista uma faixa de tamanho aceitável para pacotes de classes. Segundo tais autores, não parece ideal ter, por exemplo, um pacote com 200 classes – embora possam existir cenários de exceção. Neste caso, a identificação de uma classe dentro deste grupo estaria dificultada e sua manutenção comprometida – este defeito de projeto é conhecido como *God Package* e refere-se a pacotes que tendem a ser muito grandes e possuir classes que não interdependem, ou seja, um conjunto de classes com baixa coesão (MACÍA e STAA 2009). Desta forma, foi adicionada uma etapa pós-reorganização na técnica proposta apresentada neste trabalho. Esta etapa é responsável por verificar (e aplicar os devidos ajustes) os pacotes quanto ao seu tamanho.

Para a verificação quanto aos limites de tamanho, são definidos alguns parâmetros, como:

- Fator *Package Size* – é o número ideal de classes por pacote. Este parâmetro é utilizado quando um pacote muito grande é detectado e precisa ser quebrado em pacotes menores. Com este número é possível descobrir quantos novos pacotes devem ser criados e quantas classes em média devem ser colocadas em cada um. No contexto desse trabalho, utilizou-se o valor 7, baseado na pesquisa realizada sobre os softwares de código aberto;
- Máximo *Package Size* – a quantidade máxima aceitável de classes em um pacote. Este parâmetro é utilizado para classificar um pacote como *God Package*. Após a organização de classes em pacotes, qualquer pacote que tenha mais classes que o valor estabelecido neste parâmetro deve ser quebrado em dois ou mais pacotes. No contexto desse trabalho, utilizou-se o valor 20 – embora a literatura de Engenharia de Software não seja clara quanto ao limite ideal para um pacote, o capítulo 2 cita referências (MELTON e TEMPERO 2007, BAY 2008, LAKOS 1996) para este valor como limite máximo aceitável;
- Tentativas *Package Size* – número de vezes que a iteração "reordenar-validar" será repetida antes de aceitar a disposição resultante da reorganização. Este parâmetro é utilizado para impedir que o algoritmo entre em um *loop* infinito, caso a reorganização não consiga convergir para uma disposição que respeite os parâmetros definidos para *Package Size*. No contexto desse trabalho, utilizou-se o valor 10 – um valor arbitrário para um parâmetro que, no futuro, será objeto de testes de sensibilidade.

Além do tamanho máximo, foi considerado o tamanho mínimo para os conjuntos resultantes. Como a técnica pode gerar uma configuração onde existam pacotes com apenas uma classe, também se optou por estabelecer o tamanho mínimo de um pacote. Assim, todo pacote que possuir apenas uma classe deve ter essa classe movida para o pacote mais próximo e o pacote original deixa de existir.

Considerando as duas etapas do algoritmo – a reorganização e a validação quanto ao tamanho do pacote – foi estabelecido que a primeira irá se repetir até que a segunda seja satisfeita ou caso seja alcançado o número máximo de tentativas (parâmetro). Sempre que a configuração convergir (todas as classes estão nos pacotes mais próximos), os pacotes serão validados quanto ao seu tamanho. Caso a rotina de validação modifique algum pacote (reunindo-o a outro ou quebrando em dois ou mais), a rotina original de reorganização será reiniciada. Ao final, novamente será validado seu tamanho, e assim sucessivamente. A seguir, serão apresentados os sete passos do algoritmo de verificação do tamanho dos pacotes:

1. Primeiro são validados os pacotes quanto ao tamanho mínimo;
2. Se um único pacote possuir uma classe, esta é movida para o pacote mais próximo. Caso exista mais de um pacote com apenas uma classe, todas essas classes isoladas são movidas conjuntamente para um novo pacote;
3. Caso tenha ocorrido alguma mudança com relação ao tamanho mínimo de pacotes, o algoritmo de reorganização é reiniciado;
4. Caso o tamanho mínimo tenha sido satisfeito, é verificado cada pacote quanto ao tamanho máximo;
5. Caso seja detectado um pacote grande (tamanho maior que o parâmetro “Máximo Package Size”), são criados tantos pacotes quanto o resultado da divisão inteira do número de classes deste pacote pelo fator de tamanho ideal (“Fator Package Size”) menos um (de forma a reaproveitar o pacote sendo “quebrado”);
6. Depois de criados os novos pacotes, deve-se distribuir as classes aleatoriamente por eles, de forma que cada pacote fique com aproximadamente o mesmo número de classes. Posto que o número de novos pacotes foi calculado com base no número ideal de classes por pacote, todos os novos pacotes terminarão este passo com aproximadamente este número ideal de classes;
7. Em seguida, o algoritmo de reorganização é reiniciado.

Ao final, a nova configuração é listada (com as classes dispostas em pacotes segundo o PCC e ajustadas quanto aos tamanhos máximo e mínimo de classes por pacote). Outra informação importante que faz parte da saída do algoritmo é uma indicação de se a configuração convergiu ou não no que tange ao tamanho dos pacotes (“Convergiu Package Size”).

3.6. Considerações Finais

Este capítulo apresentou *ChangeFinder,* uma técnica que busca uma nova organização de classes em pacotes para um sistema de software de acordo com a freqüência de modificações conjuntas das classes, a partir de uma consulta a um *log* com o histórico de modificações das mesmas. No próximo capítulo, será apresentado um estudo experimental *in silico* realizado no sentido de avaliar a nova organização resultante da aplicação da técnica proposta sobre alguns sistemas de software. O estudo buscou indícios de mudanças em algumas métricas estruturais de qualidade destacadas para o software, sendo estas medidas antes e depois da aplicação da técnica de reorganização de classes.

O algoritmo da técnica proposta é uma variação de um algoritmo de *cluster analysis* conhecido como k-Médias. Para avaliar a distância entre duas classes foi definido um espaço para a sua representação através do vetor característico do perfil de modificação das mesmas – “Espaço de Classes”. A técnica procurou aplicar também conceitos relacionados ao tamanho do pacote ao buscar uma nova organização para as classes de um sistema. Os experimentos e os resultados serão descritos no capítulo 4, a seguir.

CAPÍTULO 4
APLICANDO "CHANGEFINDER"

4.1. Considerações Iniciais

Após a apresentação da técnica para reorganização de sistemas de software orientados a objetos baseada no PCC e de sua implementação através de uma variação do algoritmo de *cluster analysis* conhecido como k-Médias, um estudo experimental *in silico* (TRAVASSOS e BARROS 2003) foi planejado e executado para identificar indícios da viabilidade de aplicação desta técnica. Neste capítulo, serão apresentados o plano, os resultados e as lições aprendidas neste estudo. A estrutura utilizada para descrição do plano do estudo experimental foi inspirada em (WOHLIN *et al.* 2000) e (SHULL *et al.* 2001).

Em um estudo de viabilidade, os dados são coletados de acordo com algum planejamento experimental, embora o controle sobre todas as possíveis variáveis não seja atingido (SHULL *et al.* 2001). Este tipo de estudo empírico tem como objetivo oferecer aos pesquisadores informações que possibilitem, de alguma forma, justificar o aprimoramento contínuo das técnicas em desenvolvimento. Neste estudo, foi avaliada a utilidade da técnica de reorganização de sistemas orientados a objetos, no sentido de produzir uma configuração melhor do que a configuração corrente para os mesmos.

O estudo realizado buscou identificar se as mudanças estruturais propostas pela técnica de reorganização de classes em pacotes implicam em alguma alteração nas medidas relacionadas com a qualidade interna do software. Em última análise, buscaram-se indícios que permitissem responder às seguintes perguntas: o princípio *Common-Closure* (MARTIN 2002) foi utilizado no desenvolvimento dos softwares observados? Caso negativo, ao reorganizarmos o software com base neste princípio ocorreram modificações estruturais relevantes? Caso positivo, tais modificações trouxeram um ganho de qualidade segundo determinadas métricas extraídas a partir do código-fonte do software?

O restante deste capítulo está dividido em seis seções. Na seção 4.2 é apresentada a definição do estudo, ressaltando seus objetivos. Na seção 4.3 mostra-se o planejamento do estudo, considerando que este possa ser empacotado e replicado por outros pesquisadores. Na seção 4.4, é apresentada a execução do estudo, explicitando as características da ferramenta e do ambiente onde as observações foram feitas. Na seção 4.5, são listados os resultados da análise sobre os dados colhidos durante o experimento. Na seção 4.6 o capítulo é concluído, resumindo os resultados obtidos através do estudo.

4.2. Definição do Estudo

Objeto de Estudo: a utilização da técnica *ChangeFinder* para reorganização de softwares orientados a objetos de forma a aumentar a aderência destes softwares ao PCC.

Objetivo: identificar a viabilidade de utilização da técnica de reorganização de software orientado a objetos e verificar se ela provoca mudanças em métricas de qualidade interna do software.

Foco de Qualidade: os ganhos obtidos pela utilização da técnica proposta, medidos através de métricas estruturais coletadas a partir de um conjunto de softwares, observados antes e depois de aplicada a técnica proposta.

Perspectiva: o estudo será desenvolvido sob a ótica do pesquisador, avaliando a viabilidade de utilização da técnica de reorganização, tendo em vista um processo contínuo de melhoria da técnica proposta.

Contexto: a análise de softwares construídos usando linguagens de programação orientadas a objetos e que possuam histórico de modificação do código-fonte disponível. O estudo será conduzido no formato de teste individual sobre vários objetos.

A estrutura a seguir, baseada no método GQM (BASILI *et al.* 1994), descreve o objetivo do estudo.

Analisar *a utilização da técnica de reorganização de classes em pacotes*
Com o propósito de *caracterizar a viabilidade de uso e continuidade da pesquisa*
Referente *aos ganhos de qualidade interna obtidos por sua utilização*
Do ponto de vista do *pesquisador*
No contexto do *desenvolvimento e manutenção de softwares orientados a objetos.*

4.3. Planejamento do Estudo

Contexto Global: técnicas de reorganização de software (*refactoring*) são aplicadas uma vez que a estrutura do software tende a se degenerar ao longo do tempo, devido às modificações e evoluções sofridas por ele. Estas técnicas fazem parte da área de estudo da evolução de software, onde se procura analisar as mudanças no software ao longo do tempo e realizar tais mudanças da forma mais organizada possível. Lehman (1980), em sua Primeira Lei da Evolução de Software, declara que “um sistema de informação sendo utilizado é constantemente modificado ou se torna cada vez menos útil”. Assim, os softwares devem ser desenvolvidos de modo que possam ser alterados com relativa facilidade. Se a estrutura do software for adaptada para que as classes que são modificadas conjuntamente fiquem sempre no mesmo pacote, pode-se inferir que alterações nas mesmas sejam facilitadas, pois a probabilidade de alterarem classes de outros pacotes será reduzida;

Contexto Local: este estudo tem como intenção avaliar a viabilidade da utilização da técnica de reorganização de software *ChangeFinder* no *refactoring* de sistemas de software orientados a objetos. Os softwares utilizados no estudo devem ter seu código-fonte armazenado no repositório de um sistema de controle de versão e serão analisados de acordo com seu *log* de modificações. Para analisar as modificações sofridas, depois de aplicada a técnica, serão coletadas métricas estruturais visando detectar impactos na qualidade interna dos softwares utilizados no experimento.

Treinamento e **Participantes**: não se aplicam, pois o estudo será *in silico*, realizado sem intervenção humana, com exceção do próprio pesquisador.

Instrumentação: o estudo será feito utilizando-se um software construído para esta finalidade, o Simulador *ChangeFinder*. Este software recebe um arquivo de *log* em formato texto, extraído a partir de um repositório baseado no sistema de controle de versão CVS (ver seção 3.2). O Simulador também permite que sejam informados os demais parâmetros para a execução da técnica, como o tamanho do intervalo de tempo, as datas de início e fim para o filtro do *log*, a linguagem de programação na qual o sistema foi construído (Java, C#, C++, etc.), entre outros – ver seção 4.5. Ao final da execução do Simulador *ChangeFinder*, será listada a nova estrutura interna proposta para o software, representada como uma nova disposição de classes por pacotes, que servirá de base para os *refactorings* a serem aplicados.

Critérios: o foco de qualidade do estudo exige critérios que avaliem os ganhos potenciais proporcionados pela utilização da técnica de reorganização de classes em softwares OO de acordo com o princípio *Common-Closure*. Os ganhos obtidos pela utilização da técnica serão avaliados quantitativamente através de métricas estruturais, mais especificamente métricas de acoplamento (acoplamento de entrada e de saída) e distância para seqüência principal (DMS ou *normalized distance*) (MARTIN 2005) coletadas a partir do código-fonte original do software e a partir do código-fonte do software depois de aplicados os *refactorings* sugeridos pela técnica. Estas métricas serão coletadas através de um *plug-in* (Metrics[11]) para a IDE Eclipse.

Hipótese Nula: a hipótese nula determina que a utilização da técnica de reorganização de classes em pacotes não produz alterações nas métricas de qualidade quando aplicadas a um software orientado a objetos. De acordo com os critérios selecionados, esta hipótese se traduz na inexistência de diferenças significativas no acoplamento e na DMS do software após a reorganização das classes em pacotes, com relação à estrutura original do mesmo.

$$H_{ACO0}: \mu \text{ acoplamento do software sem técnica} = \mu \text{ acoplamento do software com técnica}$$

$$H_{DMS0}: \mu \text{ DMS do software sem técnica} = \mu \text{ DMS do software com técnica}$$

Hipótese Alternativa: determina que o software, depois de aplicada a técnica, possui características estruturais diferentes do software original, ocasionando uma diferença nos critérios de qualidade. De acordo com as métricas selecionadas, esta hipótese se traduz em um acoplamento diferente, idealmente menor, entre os componentes internos do software – o que, em tese, facilitaria a manutenção e a evolução do mesmo, bem como o reuso de seus componentes/pacotes (MELTON e TEMPERO 2007). Deseja-se também que a DMS seja menor, posto que idealmente essa medida deve estar próxima de zero.

$$H_{ACO1}: \mu \text{ acoplamento do software sem técnica} \neq \mu \text{ acoplamento do software com técnica}$$

$$H_{DMS1}: \mu \text{ DMS do software sem técnica} \neq \mu \text{ DMS do software com técnica}$$

Variáveis Independentes: as variáveis independentes do estudo são relacionadas às características do software e do projeto pelo qual ele foi construído, como o domínio de aplicação, o número de desenvolvedores envolvidos e a política de uso do sistema

[11] http://metrics.sourceforge.net/

de controle de versões que, em última análise, influenciará o tamanho do *log* do software e a freqüência com que as mudanças são observadas.

Variáveis Dependentes: as variáveis dependentes são o acoplamento[12] de entrada (*afferent coupling*), de saída (*efferent coupling*) e a DMS[13] (*distance from main sequence*). Acoplamento de Entrada (Ca) pode ser definido como o número de classes fora de um pacote que dependem de classes dentro do pacote sendo analisado. Acoplamento de Saída (Ce) pode ser definido como o número de classes dentro de um pacote que dependem de classes fora do pacote sendo analisado. A métrica de distância para a seqüência principal (DMS) mede o equilíbrio entre as taxas de abstração e instabilidade do pacote.

Análise Qualitativa: tem o objetivo de verificar se as métricas utilizadas para avaliar a nova disposição do software trouxeram uma melhora ou piora na estrutura do mesmo. A análise qualitativa será realizada através da coleta de métricas pós-reorganização.

Capacidade Aleatória: pode ser exercida na seleção dos softwares a partir de um repositório aberto como o *SourceForge*, desde que respeitados os pré-requisitos para aplicação da técnica (linguagem de programação orientada a objetos e disponibilização do *log* com o histórico de modificações).

Classificação em Bloco: o estudo será feito para todo o conjunto dos sistemas de software escolhidos. O estudo também será feito dividindo os sistemas em blocos, no sentido de identificar padrões de comportamento por grupos.

Mecanismos de Análise: o estudo proposto se classifica como um experimento de um único tratamento sobre vários objetos, onde as variáveis dependentes são representadas na escala razão[14]. A análise será feita através da comparação entre as métricas coletadas antes e depois de aplicada a técnica. Submetendo os valores referentes às métricas a um teste de Mann-Whitney[15] (uma alternativa não paramétrica

[12] Ver subseção 2.4.1.

[13] Ver subseção 2.4.2.

[14] É a mais completa e sofisticada das escalas numéricas. Ela é uma quantificação produzida a partir da identificação de um ponto zero que é fixo e absoluto, representando um ponto de nulidade, ausência e/ou mínimo. Nela, uma unidade de medida é definida em termos da diferença entre o ponto zero e uma intensidade conhecida. A partir disso, cada observação é aferida segundo a sua distância do ponto zero, distância essa expressa em unidades da medida que foi definida.

[15] http://www.statsdirect.com/help/nonparametric_methods/mwt.htm

ao teste *t* para a diferença de médias, ou seja, um teste não-paramétrico para avaliar se duas amostras de observações independentes têm a mesma média; é um dos mais conhecidos testes não-paramétricos de significância), busca-se averiguar se a hipótese nula de igualdade pode ser rejeitada para as métricas de acoplamento, bem como para a DMS.

Validade de Conclusão: a validade da conclusão de um estudo está relacionada com questões que podem afetar a habilidade de se chegar à conclusão correta sobre relações entre o tratamento e o resultado de um experimento. A força de um teste estatístico é a habilidade de revelar um padrão verdadeiro nos dados; se a força é baixa, existe um alto risco de que uma conclusão errônea possa ser alcançada. No contexto deste trabalho, pode-se destacar como uma ameaça à validade de conclusão o número reduzido de observações, pois apenas oito sistemas de software foram analisados.

Validade Interna: a validade interna de um estudo é definida como a capacidade de um novo estudo replicar o comportamento do estudo atual com os mesmos participantes e objetos com que ele foi realizado. Uma vez que não são necessários participantes, além do pesquisador, considera-se que a validade interna do estudo é alta. O fato de não haver grupo de controle, uma vez que o experimento possui um único participante, que é o próprio pesquisador, também aumenta sua validade interna. Por fim, a validade interna é alta posto que não há fator de aleatoriedade – embora os sistemas de software em si tenham sido escolhidos aleatoriamente.

Validade de Construção: a validade de construção do estudo se refere à relação entre os instrumentos e participantes do estudo e a teoria que está sendo avaliada por este, focando na generalização do resultado do experimento para o contexto da teoria base. Entretanto, uma vez que o instrumento leva em consideração uma informação (data de *check-in* da classe) que não é exatamente a mesma informação (data real de alteração da classe) que fundamenta a teoria (PCC), é possível que isto seja considerado como uma ameaça à validade de construção. Outra ameaça é a escolha das métricas (e valores para alguns parâmetros utilizados pela técnica) que podem não ser ideais uma vez que foram escolhas arbitrárias. Por outro lado, convém ressaltar que tais métricas foram escolhidas por ser o acoplamento um aspecto de qualidade caracterizável (baixo acoplamento) e por não existirem muitas alternativas equivalentes. Já a DMS foi escolhida por ter sido definida pelo próprio autor da teoria em análise (PCC). Também é importante notar que as ferramentas disponíveis geralmente citam essas métricas (acoplamento e DMS), formando certo consenso.

Validade Externa: a validade externa do estudo mede sua capacidade de refletir o mesmo comportamento em outros grupos de participantes e profissionais da indústria, ou seja, em outros grupos além daquele em que o estudo será aplicado. Acredita-se que a validade externa do estudo seja alta posto que através do Simulador *ChangeFinder*, qualquer interessado poderá, tendo acesso a um repositório compatível, realizar o mesmo experimento. Existem três tipos de interações com o tratamento: “pessoas”, “ambiente” e “tempo”. Como o experimento não necessita de outras pessoas que não o próprio pesquisador, não existe ameaça relacionada a “pessoas”. Com relação ao “ambiente” também não existe ameaça, posto que serão utilizadas tecnologias contemporâneas como linguagens de programação modernas e sistemas de controle de versão utilizados no mercado. Porém, por questões de propriedade intelectual, só será possível utilizar no experimento softwares de código aberto, o que se traduz numa ameaça à validade externa. O “tempo” também caracteriza uma ameaça à validade externa uma vez que se o experimento for repetido posteriormente, os resultados poderão ser diferentes dos observados em sua primeira execução já que a análise irá levar em conta um número maior de intervalos de tempo – o que tende a influenciar diretamente os resultados.

4.4. Execução do Estudo

4.4.1. A Ferramenta Utilizada no Estudo

O aplicativo construído para a realização do estudo permite que sejam feitas observações a partir de um *log* em formato texto, extraído de um repositório CVS. Esta ferramenta foi construída na linguagem Visual Basic .NET versão 8.0 e atualmente está preparada para tratar *logs* de sistemas de software construídos na linguagem de programação Java – trabalhos futuros podem adaptar a ferramenta para outras linguagens que implementem o paradigma OO, como C# .NET por exemplo. Java foi escolhida como primeira linguagem a ser suportada pelo aplicativo por conta do número considerável de projetos de código aberto que utilizam esta tecnologia.

A Figura 4.1 ilustra a tela principal da ferramenta Simulador *ChangeFinder*. Nesta tela, é possível especificar o arquivo que foi gerado a partir de um comando de extração de *log*, na sintaxe:

cvs log *nome-do-módulo* > log_*nome-do-arquivo*.txt

onde *nome-do-módulo* é o nome do módulo no repositório CVS do qual se deseja extrair o *log* de modificações e *nome-do-arquivo* é o nome do arquivo de texto que será criado com o retorno do comando executado.

A ferramenta emite diversos relatórios referentes ao processamento do arquivo do *log* e permite que os parâmetros utilizados na técnica sejam informados (tal como o filtro de revisões mínimas e o tamanho do intervalo de tempo). A tela principal também exibe um sumário dos indicadores calculados a partir da análise do *log* fornecido, como número de classes, *commits*, revisões, intervalos de tempo dos vetores característicos (descritos como “Momentos” na interface), bem como o número de modificações ocorridas dentro dos intervalos de tempo.

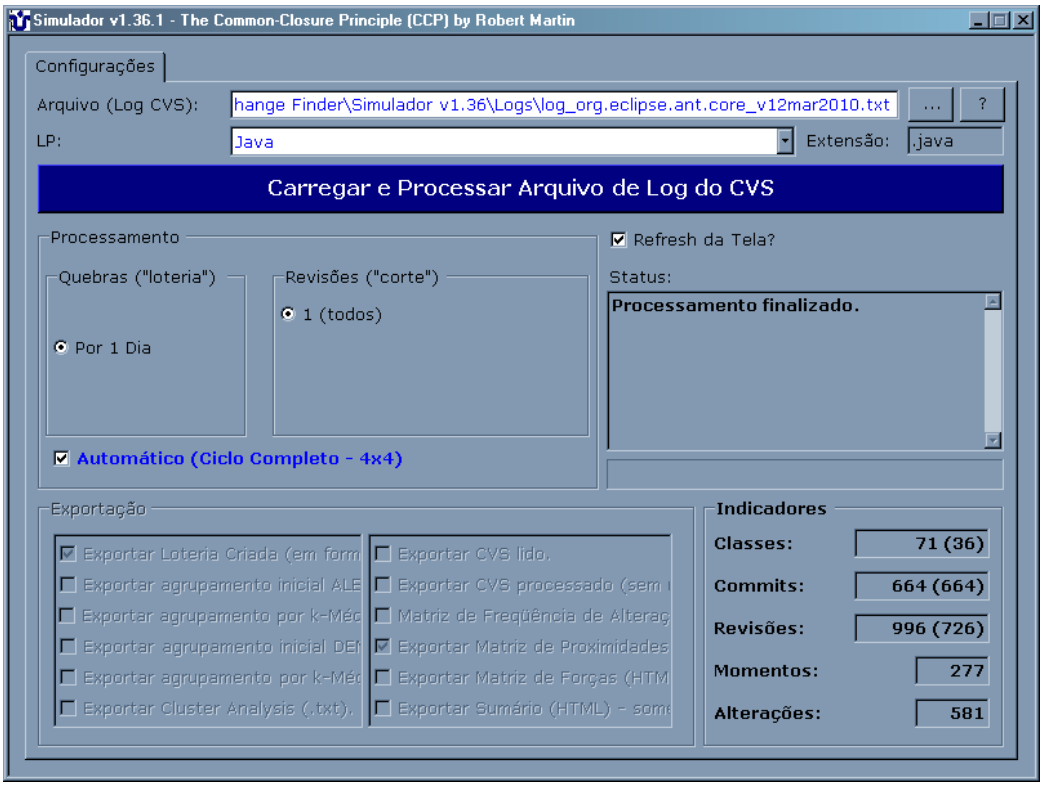

Figura 4.1. Tela principal do Simulador *ChangeFinder*

Atualmente, a etapa do experimento que mais consome tempo é o *refactoring* do projeto original para que seja possível a coleta das métricas após a aplicação da técnica de reorganização de classes em pacotes. Isto se deve ao fato de que, além do *refactoring* não ser totalmente automatizado, erros de referência podem ser gerados uma vez que a ferramenta (IDE Eclipse) não é totalmente eficaz – ao renomear pacotes ou mover classes de um pacote para outro, algumas referências ficam incorretas (*imports*), gerando erros de compilação. Realizar as modificações sugeridas pela técnica e, ao final, ter o software em um estado possível de compilação mostrou

ser uma tarefa dispendiosa e que depende consideravelmente do ferramental utilizado – o que resultou em um número pequeno de observações realizadas. Para que seja possível realizar um número maior de observações em um período menor de tempo, o simulador está sendo adaptado para utilizar também como entrada um arquivo (XML) contendo a relação de dependências entre as classes. Este arquivo é gerado a partir de um utilitário chamado CDA[16]. Ciente das dependências entre as classes, será possível calcular as métricas Ca e Ce sem a necessidade do *refactoring* em si do sistema de software – poupando tempo já que o *refactoring* só passará a ser feito se efetivamente a nova configuração for melhor que a original, de acordo com estas mesmas métricas.

O produto final da utilização do simulador é uma lista com todas as classes componentes do sistema sendo analisado, redistribuídas por pacote de acordo com o PCC. De posse desta lista, que pode ser conferida na Figura 4.2, os *refactorings* (*Move Classe* e, eventualmente, o *Rename Class* no caso de classes homônimas movidas para o mesmo pacote) devem ser aplicados sobre a configuração original do software – isto pode ser feito através de qualquer IDE que suporte a linguagem de programação do sistema de software sendo observado. No escopo deste trabalho, utilizou-se a IDE Eclipse[17] já que todas as observações do experimento foram feitas a partir de softwares construídos na linguagem de programação Java.

Vale ressaltar que o processo de nomenclatura dos "novos" pacotes foi automatizado, mas não resulta em nomes intuitivos. Ao contrário, os pacotes ficam com nomes pouco significativos, o que pode resultar em uma nova curva de aprendizado para os desenvolvedores responsáveis pela manutenção do sistema de software em questão. Posto isso, ressalta-se que a técnica proposta não busca uma solução final para o projeto do sistema de software. Na verdade, a técnica busca oferecer sugestões que podem ser aceitas, totalmente ou parcialmente, pelo arquiteto responsável pela aplicação. Tais modificações podem afetar não apenas a própria arquitetura original da aplicação (por exemplo, uma aplicação que estivesse seguindo o padrão arquitetural MVC), bem como o conhecimento dos desenvolvedores, posto que as classes componentes do sistema estarão reorganizadas em outros pacotes – bem como o próprio nome dos pacotes. Assim, recomenda-se que a nomenclatura dos novos pacotes seja considerada como um processo manual, a ser realizado durante a aplicação da técnica proposta.

[16] http://www.dependency-analyzer.org/

[17] http://www.eclipse.org/

```
> pacote_1 - Classes: 4
>> Classe AntCorePlugin.java
>> Classe AntCorePreferences.java
>> Classe AntRunner.java
>> Classe InternalAntRunner.java
------ Fim do Pacote ------
> pacote_2 - Classes: 5
>> Classe TargetInfo.java
>> Classe AntClasspathEntry.java
>> Classe IAntCoreConstants.java
>> Classe InternalCoreAntMessages.java
>> Classe InternalAntMessages.java
------ Fim do Pacote ------
*** Fim da Configuração ***
Total de Pacotes: 6
Total de Classes: 36
Classes Por Pacote: 6,000
Quantidade de 'Momentos': 277
*****************************
*** Convergiu Package Size ***
*****************************
Maior Pacote: 14
Menor Pacote: 2
```

Figura 4.2. Exemplo de listagem produzida pelo Simulador *ChangeFinder*

De posse da nova organização das classes em pacotes é desejável realizar algum tipo de validação. Esta validação tem como objetivo comparar a disposição original com a nova disposição sugerida pela técnica. A questão que se busca responder é se realmente o PCC (critério de reorganização básico da técnica) melhora o projeto do software sob algum aspecto. Para isso, foram eleitas algumas métricas que serão aplicadas antes e depois da reorganização. Também se procura verificar quão similar a configuração resultante é da configuração original, de forma a buscar indícios da utilização do PCC no desenvolvimento do software sendo observado.

4.4.2. Seleção dos Objetos de Análise

O estudo experimental foi realizado nas seguintes etapas: (i) o código-fonte e o *log* com o histórico de alterações das aplicações listadas na Tabela 4.1 foram recuperados; (ii) as aplicações (escolhidas por comodidade de acordo com sua linguagem de programação e disponibilidade do respositório CVS) foram medidas, registrando-se o número de classes, pacotes, as medidas de acoplamento e DMS; (iii) as aplicações foram submetidas à técnica proposta, obtendo-se uma sugestão de distribuição de classes por pacote; (iv) as aplicações foram refatoradas para atender à

distribuição proposta; (v) as aplicações refatoradas foram medidas novamente; e (vi) os valores coletados para as métricas de acoplamento e DMS antes e depois da refatoração foram comparados.

Tabela 4.1. Breve descrição das aplicações sob análise

Software	Breve Descrição
Azureus	Um cliente de compartilhamento P2P que utilize o protocolo bit torrent.
Eclipse ANT	Uma ferramenta de build puramente Java, apresentada como um módulo do Eclipse.
Jena	Um framework Java para construção de aplicações de Web semântica.
JMule	Um cliente de compartilhamento de código-aberto escrito em Java para redes eDonkey2000.
JUnit	Um framework Java que permite a criação de classes para teste unitário.
JVI	Uma implementação Java para o clássico editor de texto VI.
Poor Man CMS	Um CMS[18] básico sendo executado como uma aplicação Java e gerando páginas HTML estáticas.
Sweet Home 3D	Uma aplicação de projeto de interiores para escolha e posicionamento de móveis em uma planta de uma casa 2D desenhada pelo usuário final, com uma visualização 3D.

No que tange a seu projeto, estas aplicações têm as características apresentadas na Tabela 4.2. Os números de classes e de pacotes representam os números dos respectivos elementos na versão original do software, como quando baixados dos repositórios SourceForge e Eclipse. O número de intervalos representa o número de intervalos de tempo em que o *log* com o histórico de modificações da aplicação foi dividido. Um número de intervalos reduzido geralmente representa aplicações recentes, com histórico de alterações limitado, enquanto um maior número de intervalos representa aplicações mais antigas, que sofreram mais alterações ao longo do tempo.

A hipótese nula para o estudo realizado dita que o uso da técnica proposta para reorganização de classes em pacotes (e, conseqüentemente, a aplicação do princípio de projeto CCP) não gera melhorias significativas na estrutura de um sistema de software. Por outro lado, a hipótese alternativa indica que há melhora ou prejuízo na estrutura do software após a aplicação da técnica proposta. Esta melhoria será baseada em um conjunto de métricas de qualidade de projeto de software. Assim, a hipótese nula indica que não haverá diferença significativa nessas métricas, quando medidas antes e depois da redistribuição de classes em pacotes.

[18] Sistema de Gestão de Conteúdo - SGC (em inglês *Content Management System* - CMS).

Tabela 4.2. 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

4.5. Análise dos Resultados do Estudo

Os indícios utilizados para suportar as conclusões alcançadas foram baseados em métricas de acoplamento – Acoplamento de Entrada e de Saída – e características estruturais, como abstratividade e instabilidade – *Distance from the Main Sequence* – dos pacotes. Desta forma, os resultados do estudo realizado, bem como as considerações sobre os mesmos, serão apresentados em duas subseções.

As aplicações foram agrupadas de acordo com o número de classes, o número de intervalos de tempo no *log* com o histórico de modificações e a razão entre essas medidas. Em cada caso, dois grupos foram construídos: uma vez que a amostra (8 aplicações) é relativamente pequena, a construção de mais grupos poderia resultar em grupos com o máximo de duas classes.

Na próxima subseção, 4.5.1 são listados os resultados de acordo com as métricas de acoplamento. O acoplamento pode afetar tanto a facilidade de manter o software, posto que pacotes com muitas dependências tendem a ser mais difíceis de serem alterados, bem como o reuso de um módulo, já que para reutilizá-lo em outro sistema, os demais pacotes de que ele depende também precisariam ser levados.

Já na subseção 4.5.2 são listados os resultados de acordo com a métrica de distância para a seqüência principal. Esta métrica definida por Martin (1997) visa à construção de sistemas que estejam estruturados em pacotes úteis e de fácil manutenção, conforme exposto no capítulo 2.

4.5.1. Implicações no Acoplamento

As tabelas 4.3 e 4.4 apresentam a média e o desvio-padrão do acoplamento de entrada e de saída das aplicações selecionadas para o estudo experimental, antes e depois da refatoração.

Tabela 4.3. Média de valores para acoplamento de entrada e saída

Software	Ca_{ANT}	Ca_{DEP}	Ce_{ANT}	Ce_{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 4.4. Desvio padrão para acoplamento de entrada e saída

Software	Ca_{ANT}	Ca_{DEP}	Ce_{ANT}	Ce_{DEP}
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 quanto para a média. 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.

Os sistemas também foram agrupados 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 (COSTA *et al.* 2007). Para cada grupo, calculamos a variação de Ca e Ce como um percentual destes valores antes e depois da refatoração das aplicações, de acordo com as sugestões da técnica proposta. As tabelas 4.5, 4.6 e 4.7 (a razão classe/intervalo é uma medida de densidade de informação disponível sobre as mudanças sofridas pelas classes) apresentam os diferentes grupos.

Tabela 4.5. Variação de acordo com o número de classes

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

Tabela 4.6. Variação de acordo com o número de intervalos de tempo

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

Tabela 4.7. Variação de acordo com a taxa classe/intervalo

Grupos de Software (classe/intervalo)	MÉD C/I taxa	Ca Variação	Ce 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 Ca entre os grupos: o acoplamento de entrada parece aumentar sempre, independentemente do número de classes,

intervalos ou da relação entre eles. Por outro lado, a técnica proposta 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 muitos 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 (Ce representa o número de classes de quem um pacote depende para cumprir seu contrato), observou-se que a técnica de redistribuição de classes pode ser útil quando existir informação suficiente sobre as alterações sofridas pela aplicação, ou seja, quando a relação entre o número de classes e do número de intervalos for pequena.

Buscando uma relação baixa entre o número de classes e o número de registros de alteração, pode-se sugerir 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.

4.5.2. Implicações na Distância para a Seqüência Principal

A Tabela 4.8 apresenta a média e o desvio-padrão da distância para a seqüência principal para as aplicações selecionadas para o estudo experimental, antes e depois da reorganização das classes em pacotes de acordo com a técnica proposta.

Tabela 4.8. Média de valores e Desvio Padrão para DMS

Software	Média DIS_{ANT}	Média DIS_{DEP}	DP DIS_{ANT}	DP DIS_{DEP}
Azureus	0,38	0,47	0,27	0,24
Eclipse ANT	0,13	0,34	0,16	0,23
Jena	0,35	0,53	0,28	0,29
JMule	0,44	0,56	0,26	0,25
JUnit	0,36	0,52	0,26	0,23
JVI	0,40	0,55	0,35	0,19
Poor Man CMS	0,49	0,63	0,34	0,29
Sweet Home 3D	0,41	0,63	0,33	0,22

Submetendo estes valores a um teste de Mann-Whitney bicaudal usando α = 90%, a hipótese nula de igualdade não foi rejeitada para a métrica de distância para a seqüência principal, indicando que não houve mudança significativa nestes valores. Considerando-se os mesmos critérios de agrupamento utilizados para analisar os impactos sobre o acoplamento, também não se obteve diferença relevante entre os grupos. Nos três casos (agrupamento por número de classes, intervalos de tempo e relação classes por intervalos) as medidas antes e depois da aplicação da técnica apresentaram uma variação de 11% a 17%, como pode ser visto na Tabela 4.9.

Tabela 4.9. Médias (DMS) por agrupamentos

	Média DIS_{ANT}	**Média DIS_{DEP}**	**Variação**
Número de classes			
G1	0,34	0,51	+17%
G2	0,39	0,50	+11%
Número de intervalos de tempo			
G1	0,40	0,56	+16%
G2	0,35	0,51	+16%
Relação classes/intervalos			
G1	0,30	0,45	+15%
G2	0,41	0,57	+16%

Uma vez que a métrica da distância para a seqüência principal não foi afetada substancialmente, tem-se um indício de que a utilização do princípio de projeto PCC está em conformidade com esta métrica, ou seja, poderia se argumentar que existe uma relação entre o PCC e a DMS. Desta forma, considerando o que já foi exposto no capítulo 2 sobre pacotes abstratos e instáveis, pode-se considerar o PCC e a técnica proposta como uma estratégia para se obter pacotes úteis e facilmente modificáveis.

4.6. Conclusão

Neste capítulo foi apresentado o planejamento, a execução e a análise dos resultados obtidos através de um estudo experimental que buscou avaliar a viabilidade da técnica de reorganização de classes em pacotes *ChangeFinder*. O estudo foi realizado com o intuito de analisar a capacidade de um projetista reorganizando um software orientado a objetos obter uma estrutura de melhor qualidade que a estrutura atual, considerando como medida de qualidade o acoplamento entre os pacotes e a DMS dos mesmos.

Ao aplicar a técnica proposta em oito aplicações de código aberto, avaliamos se o uso do PCC poderia melhorar seus projetos em termos de acoplamento, embora os valores para DMS não tenham apresentado mudanças significativas. Observou-se que uma relação estreita entre o número de classes e o número de intervalos de tempo em que foram observadas alterações é fundamental para que a técnica proposta possa reduzir o acoplamento. Assim, a técnica proposta pode ser útil para diminuir o esforço de manutenção em sistemas de software de grande porte, que já possuam um histórico de modificações considerável. Desta forma, o estudo mostra indícios de que a técnica pode ser viável, assim como aponta algumas direções em que a pesquisa deve avançar – como passar a contemplar níveis hierárquicos, e aumentar o número de observações através da implementação do cálculo automático das métricas sem a necessidade do *refactoring* de fato.

Entretanto, repetições deste estudo, em diferentes contextos, além da realização de estudos de observação (incluindo testes de sensibilidade nos parâmetros utilizados), são ainda necessárias para a efetiva identificação de problemas e aprimoramento da técnica.

CAPÍTULO 5
CONCLUSÕES

5.1. Considerações Finais

Neste trabalho foi apresentada 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, foi analisado se o uso do PCC poderia melhorar seus projetos em termos de acoplamento e DMS. Observou-se que uma relação pequena entre o número de classes e o número de intervalos de tempo em que foram observadas as alterações é fundamental para reduzir o acoplamento, ao passo que a DMS não apresentou mudanças consideráveis. Assim, há indícios de que a técnica proposta pode ser útil para diminuir o esforço de manutenção em sistemas de software que estejam em uso por um tempo considerável.

A aplicação da técnica proposta para a reorganização das classes em pacotes de um sistema de software orientado a objetos de forma a torná-lo mais aderente ao Princípio *Common-Closure* consiste na execução das seguintes atividades: (i) identificar um sistema onde a técnica pode ser aplicada, ou seja, que tenha sido desenvolvido em uma linguagem de programação orientada a objetos e que tenha seu código-fonte gerenciado por um sistema de controle de versões com um repositório acessível; (ii) obter o *log* com o histórico de modificações dos arquivos componentes do sistema de software a partir do sistema de controle de versões; (iii) aplicar a técnica proposta de forma a obter uma nova configuração para a organização interna do sistema de software; (iv) obter, também do sistema de controle de versão, o código-fonte da versão atual do sistema de software; e (v) aplicar os *refactorings* sugeridos pela técnica de reorganização proposta, de forma a deixar o sistema de

software com a estrutura de classes em pacotes igual à proposta pela aplicação da técnica.

No contexto de manutenção de software, é importante prover um mecanismo que possibilite otimizar a estrutura do sistema de tal forma que ele possa ser mantido mais facilmente. Sistemas de software tendem a ser entrópicos, e quanto maior o período em que o mesmo sofre intervenções em seu código-fonte, maior a necessidade de se fornecer mecanismos que possam, usualmente através da implementação de técnicas de *refactoring*, compensar a eventual degradação que ocorre ao longo do tempo.

Pensando-se em manutenção e reuso, uma dimensão a ser considerada é o acoplamento entre os pacotes que compõem um sistema de software. Embora existam dois tipos de acoplamento (de entrada e de saída), parece mais natural se pensar em acoplamento de saída, uma vez que esta é uma medida da "quantidade de conhecimento" que o desenvolvedor deverá possuir para manipular determinado pacote. O acoplamento de saída indica de quantos outros pacotes um determinado pacote precisa para desempenhar sua função em um projeto de software, ou seja, pacotes que o desenvolvedor precisa considerar, conhecer suas classes, sua funcionalidade, para que consiga realizar determinada ação corretiva ou evolutiva. Se por um lado isto está muito ligado às tarefas de manutenção, por outro também gera impacto diretamente na facilidade com que o pacote poderá ser reutilizado em outro sistema de software – isto se deve ao fato do acoplamento de saída indicar os demais pacotes "consumidos" por um determinado pacote. Em última instância, tais pacotes terão que ser levados juntos numa eventual reutilização em outro sistema.

Em um processo de manutenção de software, o aplicativo *ChangeFinder* (que implementa a técnica proposta para reorganização de classes em pacotes) pode ser utilizado como ferramental para otimização periódica do sistema. O arquiteto responsável por determinado sistema pode utilizar a ferramenta de forma a analisar as sugestões oferecidas pela mesma e os ganhos obtidos com sua aplicação. Assim, a verificação e otimização do sistema de software podem ser incorporadas como uma nova etapa no processo de manutenção do software.

Os resultados obtidos no estudo experimental apontam para uma correlação entre a quantidade de informação disponível (*log* com o histórico de modificações) e resultados favoráveis à aplicação da técnica proposta. Como foi exposto no capítulo anterior, tanto nos casos de agrupamento pelo número de classes, pelo número de intervalos de tempo e pela relação entre os dois, obteve-se uma diminuição no acoplamento de saída para os sistemas de softwares que possuíam mais tempo de histórico de alterações em comparação com o número de classes compondo o

sistema. Isto faz bastante sentido posto que a técnica utiliza como insumo o perfil de modificação das classes para reagrupá-las. Sendo assim, quanto maior o número de intervalos de tempo obtidos a partir do *log* com o histórico de modificações, melhor tende a ser o agrupamento final. Com isso, têm-se indícios que a técnica é mais recomendável para sistemas que se encontram em atividade por longos períodos de tempo, mas não é muito aplicável para sistemas recém criados.

Também é importante considerar o pequeno número de observações realizadas no estudo experimental. Apesar das poucas observações, o argumento sobre o indício de melhoria é relevante posto que os sitemas observados foram escolhidos aleatoriamente. O número reduzido de observações se deve ao tempo consumido com a aplicação dos *refactorings*. Esta limitação, bem como as demais, será tratada na seção 5.3 mais adiante.

5.2. Contribuições

Podem-se destacar como principais contribuições deste trabalho:

- Identificação da necessidade, no contexto da Engenharia de Software, de se avaliar se um método para a organização de classes em pacotes é efetivamente utilizado e se os ganhos pleiteados são atingidos;
- Definição e formalização do problema de organização de classes em pacotes de acordo com o PCC;
- Definição e implementação da técnica de reorganização de classes baseada no uso de um algoritmo de clusterização adaptado do *k-Mean*, seguido de um algoritmo parametrizável para adequação a métricas de *Package Size*;
- Definição, planejamento, execução e análise dos resultados das observações para avaliar a viabilidade da abordagem proposta.

5.3. Limitações

Para avaliar a técnica proposta e as ameaças à validade do estudo experimental apresentado no capítulo anterior, observaram-se algumas limitações que se pretende abordar em futuras repetições deste estudo.

Primeiramente, a amostra de aplicações de software selecionada para análise no estudo experimental é relativamente pequena, composta por apenas 8 sistemas. Isto impõe uma ameaça à conclusão, uma vez que a força do teste estatístico (Mann-Whitney) é limitada pela pequena amostra.

Por outro lado, uma amostra focada foi selecionada, limitando as aplicações a programas Java, de código aberto, e utilizando um sistema de controle de versão específico (CVS). Assim, argumenta-se que as conclusões podem valer para tal cenário, especialmente para outros projetos de software criados ou mantidos usando uma estratégia *Open Source*.

Além disso, a técnica de distribuição de classes proposta 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. Esses fatores levam em conta como os desenvolvedores utilizam sistemas de controle de versão. 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. Em muitos projetos de software, os desenvolvedores só submetem suas alterações após o desenvolvimento completo e teste de uma funcionalidade, ou ainda depois de corrigir uma série de defeitos. 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 a capacidade de identificar as classes alteradas no mesmo intervalo de tempo, afetando a percepção de alterações concomitantes subjacente à técnica proposta. 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, não foi identificada 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, considera-se que a hipótese é sólida o suficiente para suportar a conclusão.

Em relação ao tamanho do *log* com o histórico de alterações, alguns projetos de software podem ter um *log* de tamanho considerável (referindo-se a uma década ou mais de submissões), enquanto que outros projetos podem ter iniciado o controle de versão recentemente possuindo poucas intervenções registradas (algumas submissões). No caso dos *logs* de tamanho grande, o arquiteto pode considerar que as primeiras mudanças feitas no software não refletem o perfil de mudança em curso, isto é, os tipos de alterações que o sistema tem sofrido ultimamente e que caracterizam as tendências do seu esforço de manutenção corrente. Portanto, a fim de

organizar as classes de modo a refletir o perfil de mudança atual, o arquiteto pode executar a técnica proposta considerando apenas as revisões registradas após determinada data. Neste último caso, com poucas submissões (*commits*), temos observado que o desempenho da técnica proposta é pobre. Além da técnica ser limitada pelo fato do sistema de software possuir um repositório do sistema de controle de versão acessível o suficiente para a extração do *log*, e pelo fato do sistema ter sido construído em uma linguagem de programação que suporte a estrutura *package*, é importante notar que muitos sistemas de controle de versão não tratam a operação de "renomear" adequadamente, podendo traduzir esta operação como uma exclusão seguida de nova inclusão – o que faria com que se perdesse o histórico de modificações até a data onde o arquivo foi renomeado.

Outro aspecto não considerado na atual versão da técnica, e que impõe uma limitação, é o fato de que a análise não leva em consideração o desenvolvedor que realizou a operação de *commit*. Sendo assim, dois ou mais desenvolvedores podem submeter modificações que pouco tem relação umas com as outras, e mesmo assim a técnica as consideraria como originadas pela mesma razão – já que foram submetidas dentro de um mesmo intervalo de tempo. Em ambientes onde os desenvolvedores seguem um padrão de submissão em período determinado como, por exemplo, sempre nas tardes de sexta-feira, isso pode inviabilizar a aplicação da técnica proposta. Finalmente, uma limitação da técnica de distribuição de classes em pacotes proposta é que ela não está contemplando o aspecto hierárquico dos pacotes. 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.

Apesar das limitações citadas, a solução proposta constitui-se de um ferramental com potenciais benefícios. O objetivo deste trabalho não é definir um modelo que contemple todos os cenários possíveis de desenvolvimento de software, mas sim abordar os principais aspectos de um problema relevante e recorrente na maioria das organizações e prover um ferramental capaz de sugerir bons esquemas de agrupamento de classes em pacotes, agregando benefícios substanciais aos responsáveis pela manutenção de software.

5.4. Perspectivas Futuras do Trabalho

Levando em conta o fato da pequena amostra utilizada no estudo experimental, uma possível evolução do trabalho em questão é a automação do processo de coleta de

medidas antes e depois da aplicação da técnica, de forma a reduzir o tempo despendido com a aplicação dos *refactorings* sugeridos.

Com relação à limitação que diz que alterações concomitantes realizadas por desenvolvedores diferente poderiam erroneamente indicar classes afetadas pelo mesmo eixo de mudança, uma futura evolução da técnica é passar a contemplar também o desenvolvedor que fez a submissão originando a nova revisão. Desta forma, a percepção da mudança passaria a ser vinculada ao intervalo de tempo e ao desenvolvedor. Outra possível evolução do trabalho é considerar o aspecto semântico de que o PCC fala de "mudança em conjunto" e não em "criação em conjunto". Desta forma, a técnica passaria a considerar a data de "Go Live" do sistema de software, considerando apenas as revisões subseqüentes à esta data.

Com relação à limitação relacionada à utilização do "commit" como *proxy* para as mudanças reais aplicadas às classes de um sistema de software, uma perspectiva futura do trabalho é realizar a integração com um Issue Tracking System, ou seja, um sistema de controle de solicitações de modificações. É de praxe em alguns sistemas, ao efetuar um check-in, mencionar como parte do comentário (*log*) a #ID da solicitação que motivou aquela mudança (*trouble ticket id*). Desta forma, estaria se desvinculando, tanto do intervalo de tempo quanto do desenvolvedor, as classes afetadas por um determinado eixo de mudanças – o que parece ser o ideal, já que o mesmo desenvolvedor pode submeter modificações diversas no mesmo instante, bem como uma mesma modificação pode ser realizada por diferentes desenvolvedores, e em momentos distintos.

Com relação ao aspecto não hierárquico da configuração de pacotes resultante da aplicação da técnica, pretende-se abordar esta questão, 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 (neste 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.

Outro possível trabalho futuro é analisar os impactos da aplicação da técnica sobre a visibilidade do sistema do ponto de vista dos desenvolvedores, ou seja, como a aplicação da técnica afeta os padrões arquiteturais estabelecidos, a nomenclatura dos pacotes, e o conhecimento já adquirido pelos desenvolvedores. Um estudo de caso com entrevistas aos desenvolvedores envolvidos na manutenção do sistema de software poderia buscar indícios de tais impactos.

Por fim, uma possível evolução deste trabalho é a exploração da técnica em um estudo de caso na indústria, com pessoas realmente envolvidas na manutenção do

software, e buscando obter a percepção das mesmas sobre os ganhos e perdas após a aplicação da técnica de reorganização de classes em pacotes.

REFERÊNCIAS

ASADA, T., SWONGER, R. F., BOUNDS, N. *et al.*, 1992, “The Quantified Design Space: A Tool for the Quantitative Analysis of Design”, SEI Technical Report CMU/SEI-92-TR213, Carnegie Mellon University.

BASILI, V.R., CALDIERA, G., ROMBACH, H.D., 1994, “Goal Question Metric Paradigm”, IN; Marciniak, J.J., *Encyclopedia of Software Engineering*, vol. 1, New York, NY: Wiley, pp.528 – 532

BASS, L., BUHMAN, C., COMELLA-DORDA, S., *et al.*, 2001, Volume I: *Market Assessment of Component-Based Software Engineering*, Software Engineering Institute.

BAXTER, G., FREAN, M., NOBLE, J. *et al.*, 2006, “Understanding the Shape of Java Software”, In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 397-412, Portland, Oregon, USA

BAY, J., 2008, *The ThoughtWorks Anthology – Essays on Software Technology and Innovation*, p.61. Dallas, TX, USA, The Pragmatic Bookshelf.

BEVAN, J. e WHITEHEAD Jr., E.J., 2003, "Identification of Software Instabilities," Proc. of 2003 Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada.

BEVAN, J., WHITEHEAD Jr., E. J., KIM, S., GODFREY, M., 2005, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisboa, Portugal, pp. 177-186.

BEYER, D., e NOACK, A., 2005, "Clustering Software Artifacts Based on Frequent Common Changes," Proc. of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005), St. Louis, Missouri, USA, pp. 259-268.

BLOIS, A.P.T.B., 2006, Uma Abordagem de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

BOLDRINI, J. L., COSTA, S. R., FIGUEIREDO, V. L., *et al.*, 1980, “Álgebra Linear”, 3 ed., capítulo 8, Harper & Row do Brasil.

BOOCH, G., 1987, *Software components with Ada: Structures, tools, and subsystems*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

BOOCH, G., 1991, *Object oriented design with applications*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

CAI, Y., HUYNH, S., 2007, “An Evolution Model for Software Modularity Assessment”. In: *Proceedings of the 5th International Workshop on Software Quality*, pp.3, Washington, DC, USA.

CHEESMAN, J., DANIELS, J., 2001, *UML components: a Simple Process for Specifying Component-based Software*. Addison-Wesley Longman Publishing.

CHIDAMBER, S. R., KEMERER, C. F., 1994, “A Metrics Suite for Object Oriented Design”. *IEEE Trans. Software Eng.*, vol. 20, pp. 476–493.

CHRISTENSEN, M.J., THAYER, R.H., 2002, *The Project Manager's Guide to Software Engineering's Best Practices*, IEEE Computer Society Press and John Wiley & Sons.

CLELAND-HUANG, J. e CHANG, C.K., 2003, "Event-Based Traceability for Managing Evolutionary Change", IEEE Transactions on Software Engineering (TSE), v. 29, n. 9 (September), pp. 796-810.

COAD, P. e YOURDON, E., 1991, *Object-oriented analysis*, 2nd ed. Upper Saddle River, NJ, USA, Yourdon Press.

CONSTANTINE, L.L., MYERS, G.J., STEVENS, W.P., 1974, "Structured Design", *IBM Systems Journal*, Vol. 13, No. 2, pp.115-139

COSTA, H.R., BARROS, M.O., TRAVASSOS, G.H., 2007, “Evaluating software project portfolio risks”. *Journal of Systems and Software* 80(1): 16-31

COUNSELL, S., SWIFT, S., TUCKER, A., 2005, “Object-oriented cohesion as a surrogate of software comprehension: an empirical study”. In: *Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM’05)*, Budapest, Hungary.

COX, B.J., 1986, *Object-Oriented Programming: An Evolutionary Approach*. Boston, MA, USA, Addison-Wesley.

CVS, 2010, “Manual de CVS”, Home Page disponível em http://pt.wikibooks.org/wiki/Manual_de_CVS, acessado em: 07/06/2010.

DANTAS, C., MURTA, L., WERNER, C., 2005, “Consistent Evolution of UML Models by Automatic Detection of Change Traces”, 8th International Workshop on Principles of Software Evolution, Lisboa.

DEMARCO, T., 1979, *Structured Analysis and System Specification*, Yourdon Press Computing Series, Englewood Cliff, NJ, p.310.

DIAS-NETO, A.C., 2009, *Seleção de Técnicas de Teste Baseado em Modelos*. Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

EICK, S.G., GRAVES, T.L., KARR, A.F. *et al.*, 2001, "Does Code Decay? Assessing the Evidence from Change Management Data," IEEE Transactions on Software Engineering, vol. 27, pp. 1-12.

ESTUBLIER, J., LEBLANG, D., VAN DER HOEK, A., *et al.*, 2005, "Impact of Software Engineering Research on the Practice of Software Configuration Management", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 14, n. 4 (October), pp. 1-48.

GALL, H., JAZAYERI, M., KLOESCH, R. *et al.*, 1997, “Software Evolution Observations Base don Product Release History”. In: *Proceedings of the 1997 International Conference on Software Maintenance (ICSM’97)*, Bari, Italy.

GALL, H., HAJEK, K., e JAZAYERI, M., 1998, “Detection of logical coupling based on product release history”. In Proceedings of the International Conference on Software Maintenance (ICSM 1998), pages 190–197, Bethesda, Maryland, EUA.

GALL, H., JAZAYERI, M., e KRAJEWSKI, J., 2003, “Cvs release history data for detecting logical couplings”. In Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003), pages 13–23, Helsinki, Finland.

GAMMA, E., HELM, R., JOHNSON, R. *et al.*, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, Addison-Wesley Professional Computing Series. Addison-Wesley.

GIBBS, S., TSICHRITZIS, D., CASAIS, E. *et al.*, 1990, “Class Management for Software Communities”, *Communications of the ACM*, v. 33, n. 9 (set), pp.90-103, New York, NY, USA.

GILL, N.S. e GROVER, P.S., 2003, “Component-based measurement: few useful guidelines”, ACM SIGSOFT Software Engineering Notes, Volume 28 , Issue 6, ACM New York, NY, USA.

GIT, 2010, “GIT – The fast version control system”, “Git is…”. Home Page disponível em: http://git-scm.com/. Último acesso em: 07/06/2010.

GODFREY, M. W. e ZOU, L., 2005, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," IEEE Trans. on Software Engineering, vol. 31, pp. 166- 181.

GOULÃO, M. e ABREU, F.B., 2004, “Software Components Evaluation: an Overview”, In: *Proceedings of CAPSI 2004*., Lisboa, Portugal.

GRAVES, T.L. e MOCKUS, A., 1998, "Inferring Change Effort from Configuration Management Data," Proc. of In Metrics 98: Fifth International Symposium on Software Metrics, Bethesda, Maryland, pp. 267-273.

GRAVES, T.L., KARR, A.F., MARRON, J.S., *et al.*, 2000, "Predicting Fault Incidence Using Software Change History," IEEE Transactions on Software Engineering, vol. 26, pp. 653-661.

HASS, A.M.J., 2003, *Configuration Management Principles and Practices*, Boston, MA, Pearson Education, Inc.

HOWISON, J., CROWSTON, K., 2004, "The Perils and Pitfalls of Mining Source Forge". In: *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, pp.7-11

JACOBSON, I., CHRISTERSON, M., JONSSON, P. e ÖVERGAARD, G., 1992, *Object-Oriented Software Engineering: A Use Case Driven Approach.* ACM Press. Addison-Wesley, ISBN 0201544350, pp. 69-70.

KEMERER, C.F., SLAUGHTER, S., 1999, "An Empirical Approach to Studying Software Evolution", *IEEE Transactions on Software Engineering*, v 25, nº 4, (jul).

KIM, S., PAN, K., WHITEHEAD Jr., E.J., 2005a, "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, EUA.

KIM, S., WHITEHEAD Jr., E.J., BEVAN, J., 2005b, "Analysis of Signature Change Patterns," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 64-68.

KIM, M., SAZAWAL, V., NOTKIN, D., *et al.*, 2005c, "An Empirical Study of Code Clone Genealogies," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisboa, Portugal, pp. 187-196.

KIM, S., ZIMMERMANN, T., KIM, M., et al., 2006, "TA-RE: An Exchange Language for Mining Software Repositories", MSR '06, May 22-23, Shanghai, China.

KRUCHTEN, P., 2000, *The Rational Unified Process: An Introduction, Second Edition*. United States of America, Addison-Wesley.

LAKOS, J., 1996, *Large-scale C++ software design*, p.481. Redwood City, CA, USA, Addison Wesley Longman Publishing Co. Inc.

LANZA, M. e MARINESCU, R., 2006, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer.

LEHMAN, M.M., 1980, "Programs, Life Cycles, and Laws of Software Evolution", In: *Proceedings of the IEEE*, vol.68, No.9.

LEHMAN, M.M., RAMIL, J.F., WERNICK, P.D. *et al.*, 1997, “Metrics and Laws of Software Evolution – The Nineties View”. In: *Proceedings of the Fourth Intl. Software Metrics Symposium (Metrics’97)*, pp.20, Albuquerque, NM.

LEON, A., 2000, *A Guide to Software Configuration Management*, Norwood, MA, Artech House Publishers.

LIEBERHERR, K., HOLLAND, I., RIEL, A., 1988, “Object-Oriented Programming: An Objective Sense of Style”, ACM, OOPSLA’ 88 Proceedings, Pages 323-334.

LISKOV, B.H., WING, J.M., 1994, “A Behavioral Notion of Subtyping”, ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November 1994, Pages 1811-1841.

MACÍA, I. e STAA, A.V., 2009, “Redução de Potenciais Defeitos através da Detecção de Anomalias em Diagramas de Classes” In: *Proceedings of 6th Experimental Software Engineering Latin American Workshop (ESELAW 2009)*, São Carlos, SP – Brazil

MACQUEEN, J.B., 1967, "Some Methods for classification and Analysis of Multivariate Observations”, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability", Berkeley, University of California Press, 1:281-297.

MARTIN, R.C., 1997, *Stability*, In: http://www.objectmentor.com/resources/articles/stability.pdf, acessado em 21/07/2010.

MARTIN, R.C., 2002, *Agile Software Development: Principles, Patterns, and Practices*. New Jersey, Prentice Hall.

MARTIN, R.C., 2005, “The tipping point: Stability and instability in oo design”. In: http://www.ddj.com/architect/184415285, acessado em 21/07/2010.

MASON, M., 2004, “Subversion for CVS Users by Mike Mason”. In: http://www.osdir.com/Article203.phtml, acessado em 17/12/2009.

MAYO, J., 2000, “The C# Station”, “Namespaces”. Home Page disponível em: http://www.csharp-station.com/Tutorials/Lesson06.aspx. Último acesso em: 14/03/2010.

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

MEYER, B., 1995, *Object success: a manager’s guide to object orientation, its impact on the corporation, and its use for reengineering the software process*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

MEYER, B., 1997, *Object-Oriented Software Construction*, 2d ed, p.57. Upper Saddle River, NJ, USA, Prentice-Hall, Inc.

MILLER, G. A., 1956, "The magical number seven, plus or minus two: Some limits on our capacity for processing information". *Psychological Review* 63 (2): 81–97.

MITCHELL, T., 1997, McGraw Hill, Home Page disponível em: http://www.cs.cmu.edu/~tom/mlbook.html, Último acesso em: 07/06/2010.

MOCKUS, A., FIELDING, R.F., e HERBSLEB, J., 2000, "A Case Study of Open Source Development: The Apache Server," Proc. of 22nd Int'l Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 263-272.

MOCKUS, A. e WEISS, D.M., 2001, "Globalization by Chunking: a Quantitative Approach," IEEE Software, vol. 18, pp. 30-37.

MOCKUS, A. e HERBSLEB, J., 2002, "Expertise Browser: A Quantitative Approach to Identifying Expertise," Proc. of 24rd Int'l Conference on Software Engineering (ICSE 2002), Orlando, Florida, pp. 503-512.

MOCKUS, A., ZHANG, P., e LI, P., 2005, "Drivers for Customer Perceived Software Quality," Proc. of 2005 Int'l Conference on Software Engineering (ICSE 2005), Saint Louis, Missouri, EUA.

MOLINARI, L., 2007, *Gerência de Configuração - Técnicas e Práticas no Desenvolvimento do Software*. Florianópolis. Visual Books.

MONETA, C., VERNAZZA, G., ZUNINO, R., 1990, “A Vectorial Definition of Conceptual Distance for Prototype Acquisition and Refinement”. Technical Report TUM-I9019, Technical University Munich.

MURTA, L.G.P., 2006, *Gerência De Configuração No Desenvolvimento Baseado Em Componentes*, Tese de Doutorado, Universidade Federal do Rio de Janeiro, COPPE

OMG – Object Management Group, 2004, *Unified Modeling Language (UML)*, especificação 1.5.

OMG – Object Management Group, 2007, *Unified Modeling Language* (UML), Infrastructure, especificação v2.1.1 p.158. Documento disponível em: http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF. Último acesso em: 14/03/2010.

PAGE-JONES, M., 1999, Fundamentals of Object-Oriented Design in UML, Addison-Wesley.

PRESSMAN, R.S., 1982, *Software Engineering - A Practitioner's Approach*, Fourth Edition, ISBN 0-07-052182-4.

PRESSMAN, R.S., 2004, *Software Engineering: A Practitioner's Approach*, 6th edition, McGraw-Hill.

SALEHIE, M., LI, S., TAHVILDARI, L., 2006, “A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws”. In: *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp.159-168, Washington, DC, USA.

SAMETINGER, J., 1997, *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc.

SANDLER, D., 2006, “Source control in ten minutes: a Subversion tutorial by Dan Sandler”. In: http://www.owlnet.rice.edu/~comp314/svn.html, acessado em 17/12/2009.

SARKAR, S., KAK, A.C., RAMA, G.M., 2008, “Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software”, *IEEE Transactions on Software Engineering*, v 34, nº 5, (set), pp.700-720, Piscataway, NJ, USA.

SDMETRICS, 2007, In: http://www.sdmetrics.com, acessado em 06/06/2010.

SETTIMI, R., CLELAND-HUANG, J., KHADRA, O.B., et al., 2004, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts". In: International Workshop on Principles of Software Evolution (IWPSE), pp. 49-54, Kyoto, Japão.

SHLAER, S. e MELLOR, S. J., 1992, *Object lifecycles: modeling the world in states*, Yourdon Press, Upper Saddle River, NJ, USA.

SHULL, F., CARVER, J., TRAVASSOS, G.H., 2001, “An Empirical Methodology for Introducing Software Processes”, IN: *Proceeding of the Joint 8th European Software Engineering Symposium and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria (Setembro)

SZYPERSKI, C., GRUNTZ, D. e MURER, S., 2002, *Component Software: Beyond Object-Oriented Programming*. New York, ACM Press - Addison Wesley.

TRAVASSOS, G.H. e BARROS, M.O., 2003, “Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in software Engineering”, in: *Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering*, Fraunhofer IRB Verlag, Roman Castles, Italy, pp. 117-130.

TYRON, R.C., 1939, *Cluster analysis*. New York: McGraw-Hill.

TURSKI, W.M., 1996, “Reference Model for Smooth Growth of Software Systems”, *IEEE Transactions on Software Engineering*, v 22, nº 8, (ago).

VITHARANA, P., JAIN, H., ZAHEDI, F., 2004, “Strategy-Based Design of Reusable Business Components”. *IEEE Trans. on Systems, Man and Cybernetics*, vol 34, pp. 460 – 474.

WANGENHEIM, A.V., 2006, “Análise de Agrupamentos”, Home Page disponível em: http://www.inf.ufsc.br/~patrec/agrupamentos.html. Último acesso em: 07/06/2010.

WERNER, C., VASCONCELOS, A., MOURA, A.M., 2008, “Uma Estratégia de Reestruturação de Modelos baseada em Métricas para Apoiar a Reengenharia de Software Orientado a Objetos para Componentes”. In: *Proceeding of the II Brazilian Symposium on Software Components, Architectures, and Reuse*, Porto Alegre, Brazil.

WILLIAMS, J. D., 2000, "Raising Components." *Application Development Trends* 7(9). pp.27-32.

WOHLIN, C., RUNESON, P., HÖST, M. et al., 2000, *Experimentation in Software Engineering: an Introduction*, Norwell, MA: Kluver Academic Publishers

XAVIER, J. R., WERNER, C.M.L., TRAVASSOS, G.H., 2002, “Uma Abordagem para a Seleção de Padrões Arquiteturais Baseada em Características de Qualidade”, XVI Simpósio Brasileiro de Engenharia de Software, Gramado, RS, Brasil.

ZIMMERMANN, T., DIEHL, S., e ZELLER, A., 2003, “How history justifies system architecture (or not)”. In IWPSE ’03: Proceedings of the 6th International Workshop on Principles of Software Evolution, pages 73–84, Helsinki, Finland.

ZIMMERMANN, T., WEIßGERBER, P., DIEHL, S. *et al.*, 2005, "Mining Version Histories to Guide Software Changes," IEEE Trans. Software Engineering, vol. 31, pp. 429-445.

ZIMMERMANN, T., KIM, S., ZELLER, A. *et al.*, 2006, “Mining Version Archives for Co-changed Lines”, MSR’06, May 22–23, 2006, Shanghai, China.