Uma Abordagem Para Reestruturação de Sistemas de Software Orientados a Objetos

Danilo Santos, Paulo Afonso Júnior, Heitor Costa

Departamento de Ciência da Computação - Universidade Federal de Lavras - MG - Brasil

danilobatista@posgrad.ufla.br, {pauloa.junior, heitor}@dcc.ufla.br

Resumo. *A qualidade de software é uma das principais preocupações desde as fases iniciais de desenvolvimento do software. Para isso, o software deve evoluir constantemente para atender as necessidades dos usuários e do ambiente (lei da mudança contínua). Por isso, são realizadas manutenções no software. Mas, se as manutenções forem conduzidas em discordância aos padrões de projeto empregados e às melhores práticas de programação, o software se tornará progressivamente mais acoplado e menos coeso, degradando sua qualidade, pois se torna menos modularizado. Assim, neste artigo, é proposta uma abordagem para reestruturar sistemas de software baseada na movimentação de classes entre pacotes para aprimorar a modularização e a manutenibilidade desses sistemas, melhorando sua qualidade estrutural. A movimentação de classes é determinada por uma heurística de otimização combinatória (Simulated Annealing). O resultado da avaliação indica que essa abordagem aprimora simultaneamente o acoplamento e a coesão do software, movimentando baixo percentual de classes, sem afetar estrutura de pacotes do software.*

Abstract. *Software quality is a main concern since initial phases of software development. Thus, software must involute constantly for supporting users' and environment needs (continuous change law). Hence, software maintenances are carried out. But, if those maintenances are conducted in disagree to the design patterns used and to the best programming practices, software become progressively more coupled and less cohesive; it degrades the quality software, because software become less modularized. This paper shows an approach for restructuring of software by moving classes between packets to improve its modularization and maintainability; consequently, structural quality is also improved. The moving of classes is determined by a combinatorial optimizing heuristic (Simulated Annealing). The results of evaluation indicate this approach improves coupling and cohesion, simultaneously, moving few classes without affecting packet structure of software.*

1. Introdução

Sistemas de software são projetados para possuírem alto nível de qualidade desde suas fases iniciais, para serem competitivos no mercado de software. Mas, o ambiente no qual esses sistemas estão inseridos é altamente mutável. Isso deve-se a diversos fatores, como surgimento de novas tecnologias e de novas regras de negócio. Dessa maneira, para satisfazer continuamente as necessidades do ambiente e dos usuários, esses sistemas devem evoluir constantemente, *lei da mudança contínua* [Lehman, 1980]. Sendo assim, o software deve passar por constantes manutenções para corrigir, adicionar ou modificar sua funcionalidade. Isso contribui para o sistema não se tornar obsoleto e ser "abandonado" em detrimento de um concorrente que atenda melhor as necessidades

existentes [Kan, 2002]. Entretanto, não basta realizar manutenção. Caso a manutenção seja executada em discordância com padrões de projeto empregados e boas práticas de desenvolvimento preconizadas pela Engenharia de Software, gerar-se degradação na qualidade estrutural do software [Bianchi *et al.*, 2001] (também chamada de erosão de software [Gurp; Bosch, 2002]). Esse fato pode ser compreendido pela *lei do aumento da complexidade* [Lehman, 1980], a qual relata que a complexidade do software aumentará, a menos que um esforço seja realizado para reduzi-la. Essa degradação traz diversas consequências negativas ao software, como a geração de código desorganizado, ilegível e propenso a falhas [Erdil *et al.*, 2003]. Consequentemente, a degradação reduz a manutenibilidade do software [Bianchi *et al.*, 2001], aumentando acoplamento e reduzindo coesão [Lanza; Marinescu, 2006].

Tendo em vista o aumento do acoplamento e a redução da coesão, o estado da degradação estrutural do software pode ser conhecido ao longo do seu ciclo de vida por meio das medidas de software [Honglei *et al.*, 2009]. Esse conhecimento propicia um norte para aplicar reestruturações sobre o software, pois existe estreito relacionamento entre as medidas de software e a reestruturação. Esse relacionamento deve-se ao fato de, por meio das medidas, analisar o software e determinar entidades (*i.e.*, pacotes, classes e métodos) que apresentam valores "ruins" em relação a uma medida. Portanto, essas entidades podem ser reestruturadas, movimentando-as para outro local na estrutura do software para aprimorar o valor da medida analisada. Por exemplo, analisando um software `S` hipotético com `50 classes`, tem-se uma classe com a medida `X` igual a `100`, enquanto as demais classes apresentam valor máximo igual a `10`; logo, uma análise minuciosa sobre essa classe deve ser realizada, pois ela apresenta valor discrepante para a medida `X`, que pode representar uma possibilidade de reestruturação. Desse modo, as medidas encontram o ponto em que a reestruturação deve agir para gerar ganhos à qualidade estrutural do software, superando o desafio da reestruturação, que se refere a encontrar oportunidades para realizar a reestruturação [Du Bois *et al.*, 2004].

Com isso, a reestruturação agirá na contracorrente da degradação da qualidade do software, pois a reestruturação gerará um software menos acoplado e mais coeso [Du Bois *et al.*, 2004], ou seja, mais modularizado. Por consequência, esse software também será mais manutenível; logo, obterá redução nos recursos gastos na condução de futuras manutenções [Swanson, 1999]. Há alguns trabalhos que trouxeram à luz abordagens baseadas na reestruturação de software para reduzir os impactos da degradação da qualidade do software [Zanetti *et al.*, 2014; Pinto; Costa, 2014; Palomba *et al.*, 2015]. Tais abordagens prezam pelo mesmo objetivo, aprimorar a qualidade estrutural do software, buscando a construção de pacotes mais coesos e menos acoplados. Para alcançar esse objetivo, as abordagens propostas utilizam diferentes técnicas de reestruturação, tais como, movimentação de classes entre pacotes, divisão e união de pacotes. Além disso, utilizam o apoio de medidas de software para guiar a reestruturação. Porém, os trabalhos relacionados apresentam algumas limitações, por exemplo:

- Escolher a classe a ser movimentada de forma *ad hoc*, o que pode acarretar extenso consumo de tempo e outros recursos, além de haver a possibilidade de desconsiderar classes que poderiam gerar ganhos à estrutura, visto que as classes são escolhidas aleatoriamente;
- Basear-se na avaliação de todas as possibilidades de movimento de classes entre pacote para obter uma estrutura "melhor" (força bruta) fato que, apesar de encontrar a melhor solução, pode ser inviável para sistemas de médio e grande porte, por consumir muitos recursos, por exemplo, tempo de processamento.

Com base nessas limitações, neste artigo, uma abordagem de reestruturação determinística para sistemas de software é proposta. Essa abordagem possui um conjunto de passos definidos para escolher a "melhor" classe a ser movimentada, com o respaldo de medidas de software ratificadas na academia, aprimora simultaneamente o acoplamento e a coesão e utiliza a heurística de otimização combinatória *Simulated Annealing*. Essa abordagem está baseada na movimentação de classes entre pacotes, para alcançar melhor modularização e manutenibilidade do software. Para aumentar a probabilidade dessa abordagem ser aplicada na prática pelo mercado, um *plug-in* para o Eclipse IDE foi desenvolvido.

O restante deste trabalho está organizado da seguinte forma. Alguns conceitos para compreender este trabalho são apresentados na Seção 2. Os trabalhos relacionados são elencados na Seção 3. A abordagem de reestruturação é explicada na Seção 4. A avaliação dessa abordagem e os resultados são discutidos na Seção 5. As limitações e ameaças à validade são elencadas na Seção 6. Lições aprendidas são destacadas na Seção 7. Conclusões e sugestões de trabalhos futuros são discutidos na Seção 8.

2. Referencial Teórico

Nesta seção, são apresentados brevemente conceitos básicos relacionados aos assuntos utilizados para realizar este trabalho.

2.1. Manutenção

A manutenção refere-se ao processo de modificar o software após sua entrega ao cliente para corrigir falhas (manutenção corretiva), adicionar funções (manutenção evolutiva) e/ou adaptá-lo ao ambiente ou a outros atributos (manutenção adaptativa) [Erdil *et al*., 2003; Sommerville, 2010; Pressman; Maxim, 2014]. Porém, independentemente do tipo da manutenção, a manutenção é a etapa mais onerosa no ciclo de vida do software. Mesmo não havendo consenso, pode-se afirmar que essa etapa consome entre 70% e 90% dos recursos do projeto [Erdil *et al*., 2003; Erlikh, 2000; Pressman; Maxim, 2014]. O esforço atrelado à execução da manutenção também se destaca, sendo a etapa que exige mais esforço dentre as relacionadas à Engenharia de Software [Sneed; Brössler, 2003]. Esses fatos devem-se à difícil previsão e controle de execução da manutenção, cuja dificuldade é crescente à medida que a complexidade do software aumenta [Bhatt *et al*., 2004].

2.2. Medidas de Software

Medidas de Software são variáveis as quais um valor é associado como resultado do processo de medição [ISO/IEC 25000, 2014]. Essas medidas são elementos-chave em qualquer processo de engenharia [Pressman; Maxim, 2014], não sendo diferente na área Engenharia de Software. Nessa área, as medidas propiciam meios de avaliar quantitativamente características/atributos de sistemas de software [Honglei *et al.*, 2009]. Dessa forma, as medidas fornecem bases sólidas para a tomada de decisão dos Engenheiros de Software [Gyimothy, 2009], aumentando seu entendimento sobre o processo, o software e o ambiente no qual ele está inserido. Assim, as medidas tornam-se facilitadores para a gerência e o aprimoramento da qualidade do software [Focus, 2015].

Com base em um conjunto de medidas obtido em uma Revisão Sistemática da Literatura [Santos *et al.*, 2016], sete medidas de software foram escolhidas para serem utilizadas nesta investigação de acordo com os seguintes critérios: i) serem medidas difundidas na academia; ii) terem artigos recomendando-as para medir acoplamento e

coesão; e iii) serem medidas relacionadas à manutenibilidade de sistemas de software ou adequarem-se às necessidades deste trabalho. As medidas escolhidas foram:

- **MPC** (*Message Passing Coupling*). Seu objetivo é analisar a complexidade da comunicação entre classes [Li; Henry, 1993]. Seu valor é determinado pela quantidade de chamadas a métodos definidos em outras classes, indicando o quanto os métodos locais são dependentes de métodos de outras classes;
- **CBO** (*Coupling Between Objects*). Seu objetivo é indicar o quanto a classe está acoplada ao restante do software. Seu valor é contabilizado pela quantidade de classes acopladas à classe analisada, considerando como acoplamento quando uma classe possui métodos que utiliza métodos ou instancia variáveis de outra classe [Chidamber; Kemerer, 1994];
- **RFC** (*Response for Class*). Seu objetivo é indicar o potencial de comunicação da classe com as demais classes do software. Seu valor é mensurado pela quantidade de métodos da classe que podem ser acionados em resposta a uma mensagem recebida [Chidamber; Kemerer, 1994];
- **Ca** (*Afferent Coupling*). Seu objetivo é analisar a responsabilidade do pacote, informando quais classes dependem dele [Martin, 1994]. Seu valor é obtido pela quantidade de classes fora do pacote analisado com dependências incidentes sobre ele;
- **Ce** (*Efferent Coupling*). Seu objetivo é analisar o grau de dependência de um pacote em relação as demais partes do sistema [Martin, 1994]. Seu valor é contabilizado pela quantidade de classes internas ao pacote que dependem de classes externas a esse pacote;
- **LCOM4** (*Lack of Cohesion in Methods*). Essa medida é a evolução das versões anteriores (LCOM1, LCOM2 e LCOM3). Seu objetivo é avaliar a falta de coesão [Hitz; Montazeri, 1995] e baseia-se no princípio de grafo, cujos vértices são métodos e arestas são dependências que um método possui sobre outros métodos, contabilizando como dependência métodos utilizados ou atributos instanciados, direta ou indiretamente. Seu valor é determinado pela quantidade de grafos desconexos existentes na classe;
- **TCC** (*Tight Class Cohesion*). Seu objetivo é avaliar a quantidade de métodos diretamente conectados a classe, quando dois métodos utilizam variáveis em comum [Bieman *et al.*, 1995]. Seu valor é determinado pela fórmula: $TCC(C) = \frac{NDC(C)}{NP(C)}$, sendo C a classe analisada, NDC a quantidade de conexões diretas à classe e NP o total de conexões diretas e indiretas a classe. O valor de NP é determinado pela fórmula: $NP(C) = \frac{N*(N-1)}{2}$, sendo N a quantidade de métodos existentes na classe analisada.

2.3. Reestruturação

Reestruturação de software é um processo para realizar alterações na arquitetura de sistemas de software [Arnold, 1989] para reduzir os impactos negativos ocasionados pela degradação desses sistemas e aprimorar sua qualidade. Em geral, a reestruturação realiza alterações simples na estrutura do software, tendo como ideia essencial realizar redistribuições de entidades ao longo da hierarquia do software. Dessa maneira, a reestruturação só age em características não funcionais (e.g., manutenibilidade). Portanto, a reestruturação visa melhorar a arquitetura do software, transformando sua representação atual em outra no mesmo nível de abstração, sem interferir no seu comportamento perante ao usuário [OpdyKe, 1992]. Como resultado, a reestruturação pode tornar o software mais

fácil de entender e de modificar e menos susceptível a erros em futuras manutenções [Arnold, 1989].

2.4. *Simulated Annealing*

A heurística de otimização combinatória *Simulated Annealing* (SA) [Kirkpatrick, 1983] consiste na resolução de problemas de otimização combinatório, em que se busca maximizar ou minimizar o custo de uma função dentro de um espaço de busca. A otimização realizada pela SA é feita em etapas, baseando-se na busca de vizinhos, em que, a cada etapa, uma solução sugerida é gerada, a partir de alterações controladas ou aleatórias, na solução atual em busca de estados próximos que possuam estrutura melhor, ou seja, que o valor da função objetivo seja menor que o apresentado pela solução atual. Posteriormente, é verificado se a solução sugerida é melhor que a atual. Em caso positivo, a solução sugerida torna-se a solução atual; em caso negativo, a SA verifica a possibilidade aceitar uma solução pior. Assim, busca-se evitar mínimos locais, em que se acredita estar na melhor solução possível (mínimo global), mas, na realidade, trata-se de um ponto que somente possui um valor melhor do que os apresentados pelos seus vizinhos mais próximos (mínimos locais). Logo, para evitar a convergência a mínimos locais a SA pode aceitar uma solução cujo valor da função objetivo é pior que o atual. Realizada essa decisão, a SA executará até que o menor ou um dos menores valores da função objetivo do espaço de busca seja encontrado.

3. Trabalhos Relacionados

Alguns trabalhos dedicaram esforços para avançar a fronteira do conhecimento em relação a melhoria da qualidade estrutural de software por meio da reestruturação. Em um trabalho, uma abordagem para decompor o pacote em unidades menores e mais coesas foi proposta [Palomba *et al.*, 2015]. Essa abordagem realiza análise dos relacionamentos das classes do software, com base nas medidas ICP (*Information Flow-Based Coupling*) [Lee, 1995] e CCBC (*Conceptual Coupling Between Classes*). Posteriormente, baseado nessa análise e seguindo as restrições do usuário, essa abordagem reorganiza as classes do pacote escolhido em unidades menores e mais coesas. O resultado da avaliação dessa abordagem revela sua capacidade de aprimorar a qualidade do software ao gerar pacotes com melhor coesão e sem deteriorar seu acoplamento.

Em outro trabalho [Zanetti *et al.*, 2014], foi apresentada uma estratégia para agir na contracorrente da deterioração da qualidade de software. Seu objetivo é aprimorar a modularização do software movimentando classes entre pacotes. Para isso, uma classe do software é escolhida aleatoriamente e é calculada a probabilidade de movimentá-la para pacotes aos quais ela está conectada. Essa abordagem foi avaliada com um estudo de caso com 39 sistemas de software, obtendo em média 77% de aperfeiçoamento na modularização dos sistemas em relação as suas estruturas originais. Além disso, foram comparadas as opções de movimentação sugeridas pela estratégia proposta e as sugeridas por especialistas em reestruturação. O resultado revelou baixo valor de *Precision* e alto valor de *Recall*, ou seja, divergência entre o apresentado pelos especialistas e o atingido pela estratégia desenvolvida nesse trabalho.

No terceiro trabalho [Pinto; Costa, 2014], o objetivo foi reestruturar sistemas de software por meio da movimentação de classes entre pacotes, embasando-se na avaliação de todas as possibilidades de organização estrutural existentes (força bruta) e nas medidas de CBO e LCOM (*Lack of Cohesion in Methods*) [Chidamber; Kemerer, 1994]. Essa abordagem avalia se a coesão de uma classe em um pacote é menor que seu acoplamento com outro pacote. Em caso positivo, a classe é movimentada para o pacote ao qual ela

está mais acoplada, realizando esse processo para todo o sistema de software. Para avaliar, foi realizado um estudo de caso em cinco sistemas de software. Os resultados indicam melhoria na modularização do software utilizando a abordagem proposta.

Assim como os trabalhos citados, esta investigação aprimora a qualidade de sistemas de software por meio da reestruturação. Mas, é realizada a reestruturação do software com base em uma abordagem determinística, que movimenta classes entre pacotes para aumentar o acoplamento e diminuir a coesão, aprimorando a modularização e a manutenibilidade desses sistemas. Além disso, a reestruturação proposta não afeta a estrutura original dos pacotes do software, pois não adiciona, divide ou remove pacotes; assim, gera-se menor impacto à estrutura atual do software, tornando sua estrutura sugerida mais próxima da atual e mais compreensível à equipe responsável por esse software. Outras diferenças dessa abordagem são utilizar medidas de software amplamente difundidas na academia e a heurística de otimização combinatória *Simulated Annealing* para realizar a reestruturação.

4. Abordagem de Reestruturação

A abordagem de reestruturação proposta neste trabalho é composta de sete atividades (Figura 1) e objetiva aprimorar a qualidade estrutural do software por meio da movimentação de classes entre pacotes. Para avaliar se esse objetivo foi atingido, são utilizadas medidas de software [Bryton; Abreu, 2008]. Por isso, em **1** (Figura 1), o software é medido por sete medidas de software, divididas em dois grupos: i) **Grupo de Avaliação**, avalia se os ganhos proporcionados à estrutura do software são consistentes e aprimoraram a qualidade do software, sendo composto pelas medidas Ca, Ce e LCOM4; e ii) **Grupo de Reestruturação**, guia o processo de reestruturação, indicando se uma estrutura obtida durante a reestruturação é melhor/pior que a solução atual, sendo composto pelas medidas MPC, CBO, RFC e TCC. As medidas que compõem esses grupos são distintas para a avaliação da reestruturação ser imparcial ao processo de condução. Assim, pretende-se evitar vieses na avaliação, pois, ao conduzir o processo de reestruturação com uma medida e avaliá-lo com essa medida, certamente serão identificadas melhorias.

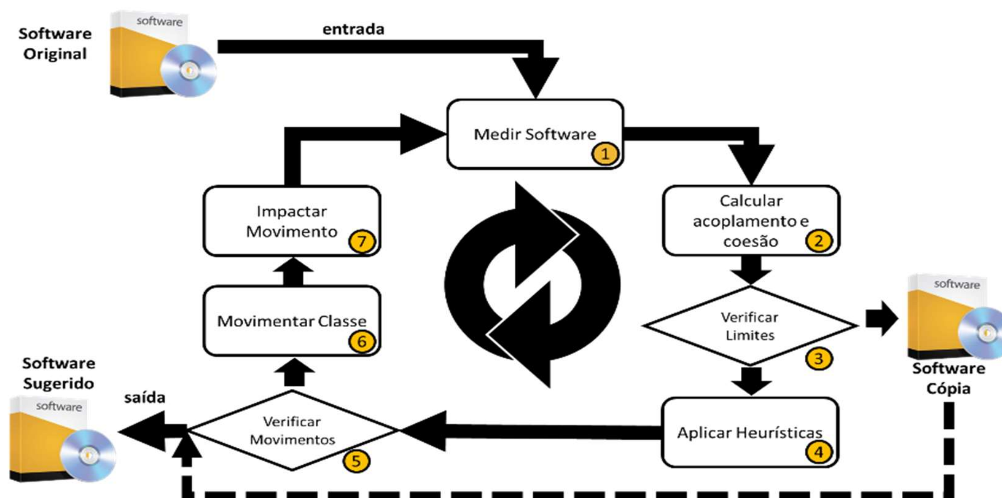

Figura 1 - Abordagem de Reestruturação Proposta

Para serem utilizadas nesta investigação, a granularidade de medição das medidas escolhidas foi alterada em relação as suas granularidades originais, sendo alteradas da granularidade de classe para a de pacotes (exceto as medidas Ca e Ce que originalmente

são aferidas nessa granularidade). Ressalta-se que essas medidas seguem suas definições originais, mas contabilizando relacionamentos entre pacotes. Essa alteração é necessária pelo fato que, ao movimentar classes entre pacotes, se a medição for realizada na granularidade de classes, os valores das medidas permanecem constantes, pois suas dependências mantêm-se inalteradas, somente deixando de ser interna para tornar-se externa ao pacote ou vice-versa. Dessa maneira, para detectar se o movimento realizado melhorou a estrutura do software, a medição deve ser conduzida na granularidade de pacotes, pois é nessa granularidade que as dependências se alteram como resultado dos movimentos de classes na abordagem proposta. Para exemplificar essa alteração na granularidade, a `Classe D` possui mais dependências com o `Pacote1` do que com seu pacote atual (Figura 2a). Assim, é justificada a movimentação da `Classe D` para o `Pacote1` (Figura 2b).

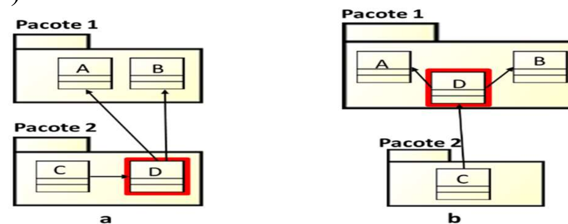

Figura 2 - a) Exemplo da Alteração na Granularidade das Medidas, antes do Movimento; b) Exemplo da Alteração na Granularidade das Medidas, depois do Movimento

A avaliação da quantidade de dependências da classe pode ser realizada para averiguar se o movimento gerou ou não consequências positivas. Avaliando as dependências na granularidade de classes, nota-se que a `Classe D` possui `3` dependências antes e após a efetivação do movimento (`Classe C`, `Classe A` e `Classe B`). Nessa situação, a modularização visualmente foi aprimorada (Figura 2b), porém essa melhoria não foi detectada quantitativamente. Avaliando as dependências na granularidade de pacotes, percebe-se que a quantidade de dependências entre pacotes da `Classe D` foi reduzida de `2` dependências (`Classe A` e `Classe B`) para `1` dependência (`Classe C`) com o movimento realizado. Portanto, com essa alteração na granularidade de medição, o aprimoramento proporcionado à modularização é perceptível visualmente e quantitativamente.

As medidas que tiveram sua granularidade de medição alterada receberam o índice `p`, para ressaltar essa alteração. A seguir, é apresentada a descrição de como calculá-las:

- **MPC_p**. Seu valor é obtido pela quantidade de métodos que a classe analisada chama de classes externas ao seu pacote. Dessa forma, pode-se saber o quanto a classe analisada é dependente de outros pacotes. Uma alta dependência indica que a classe possui muitas chamadas fora do seu pacote e pode estar mal posicionada na estrutura;
- **CBO_p**. Seu valor é obtido pela quantidade de classes sobre as quais a classe analisada possui dependências fora do seu pacote, chama métodos ou instancia atributos. Assim, pode-se saber o quanto uma classe está acoplada aos demais pacotes. Quanto maior esse acoplamento, pior, pois a dependência de muitas classes fora do seu pacote pode ser um indicativo que a classe está mal posicionada na estrutura;
- **RFC_p**. Seu valor é obtido pela soma da quantidade de métodos existentes na classe analisada mais a quantidade de métodos chamados por essa classe fora do seu pacote. Essa medida informa o nível de comunicação que uma classe tem com os pacotes. Quanto maior seu valor, maior a interação da classe analisada com outros pacotes e maior a probabilidade dessa classe estar em um pacote inadequado;
- **$LCOM4_p$**. Ao contrário das demais medidas, $LCOM4_p$ é contabilizado para cada

pacote e não para cada classe, visto que se deseja saber a coesão do pacote e não da classe. Para obter o valor de $LCOM4_p$ para o pacote são construídos grafos com base nas dependências entre as classes que o compõem, considerando métodos chamados e atributos instanciados. Posteriormente, com base na quantidade de grafos desconexos obtidos, determina-se o valor de $LCOM4_p$ para o pacote. Dessa maneira, é determinado o quão coeso é cada pacote e a redução do valor de $LCOM4_p$ indica que os pacotes do software estão mais coesos, pois possuem menos conjuntos de classes que não se relacionam com as demais classes do pacote, formando grafos desconexos;

- **TCC_p**. O valor dessa medida é obtido com base na sua fórmula original, $TCC(C) = \frac{NDC(C)}{NP(C)}$, sendo $NDC(C)$ a quantidade de métodos que chamam métodos ou instanciam atributos da classe analisada dentro do pacote da classe analisada, desconsiderando relacionamentos entre pacotes, NP a sua fórmula padrão, $NP(C) = \frac{N*(N-1)}{2}$, sendo N a quantidade de métodos existentes na classe analisada ($QuantM_{ca}$). Dessa maneira, sabe-se o quão forte é o relacionamento da classe analisada com as demais classes do seu pacote, em que o aumento do valor de TCC_p indica que esse relacionamento foi fortalecido, pois a classe aumentou seus relacionamentos com classes do seu pacote.

Seguindo as atividades da abordagem de reestruturação proposta, em **2** (Figura 1) com base nos valores das medidas do grupo de reestruturação, são calculados os níveis de acoplamento e de coesão do software analisado, por meio das seguintes fórmulas:

$$\boldsymbol{acoplamento} = CBO_p + RFC_p + MPC_p \qquad \boldsymbol{coess\tilde{a}o} = TCC_p$$

Para obter o valor do acoplamento, foi utilizada a soma dos valores das medidas CBO_p, RFC_p e MPC_p, pois essas medidas aferem o mesmo atributo sob pontos de vista diferentes, possuem mesmo sentido de crescimento, não possuem valores inversos e apresentam correlação [Barbosa; Hirama, 2013; Lima; Resende, 2014; Silva; Costa 2015]; logo, se o valor de uma medida aumenta ou reduz, mesmo que em escalas diferentes, o valor das demais tende a seguir o mesmo comportamento. Dessa maneira é possível constatar melhoria/piora no estado do sistema pela observação das somas desses valores. Sendo assim, quanto maior o valor do acoplamento, pior é a estrutura do sistema. Outro ponto importante é não adotar pesos para as medidas utilizadas para não priorizar uma medida em relação as demais.

Com isso, em **3** (Figura 1), são verificados se os limites definidos são respeitados. Esses limites são definidos previamente pelo usuário para determinar a porcentagem máxima de deterioração aceitável em um atributo em prol do aprimoramento em outro atributo. Por exemplo, caso o usuário defina que essa porcentagem é 30% para acoplamento e 20% para coesão e os valores iniciais desses atributos são respectivamente 100 e 100, então os limites a serem respeitados são 130 e 80, respectivamente. Dessa forma, caso algum dos limites definidos sejam violados, uma cópia da última estrutura que os supra é armazenada e a reestruturação segue seu fluxo normal. Sendo assim, em **4** (Figura 1), são utilizadas heurísticas do grupo de movimentação para determinar a probabilidade de movimentar cada classe do software (***PMov***). Tais heurísticas são o cerne dessa abordagem de reestruturação, pois elas encontram oportunidades de reestruturação, fato que não é uma tarefa trivial [Du Bois *et al.*, 2004]. As heurísticas que compõem esse grupo são:

- `Heurística de Movimentação I (HMI) - não mover classes de pacotes que contenham somente uma classe.` Seu objetivo é evitar o movimento da única classe de um pacote para outra parte do projeto, ocasionando a existência de um pacote vazio e consequente necessidade de excluí-lo. Sua existência justifica-se pela necessidade de não afetar a estrutura de pacotes existente no software,

em que a exclusão de pacotes geraria maior modificação na estrutura do software, dificultando a posterior compreensão da estrutura sugerida pelos mantenedores. Por isso, ao utilizar essa heurística, classes únicas em um pacote recebem a $PMov$ 0 (zero), evitando que pacotes sejam excluídos da estrutura corrente do projeto. Essa heurística pode ser expressa pela seguinte expressão, sendo P_i o conjunto de classes do pacote *i*;

$$\boldsymbol{Heurística\ I}:\ \forall P_i \mid |P_i| = 1, então\ PMov = 0$$

- Heurística de Movimentação II (HMII) - obter probabilidade de a classe ser movimentada para um pacote vizinho. Nesse contexto, pacote vizinho é um pacote que possui classes que se relacionam com a classe analisada. Essa heurística é utilizada para obter a $PMov$ das classes não abrangidas por HMI. Para obter essa probabilidade, é utilizada, como base para o cálculo, a quantidade de dependências que a classe analisada apresenta com cada pacote vizinho, sendo considerado como dependências chamadas a métodos e instanciação de variáveis. Dessa maneira, *PMov* pode ser definida pela seguinte fórmula:

  $$\boldsymbol{PMov} = DepExterno - DepInterno$$

  sendo $DepExterno$ e $DepInterno$ a quantidade de dependências que a classe apresenta fora do seu pacote e $DepInterno$ a quantidade de dependências que a classe apresenta dento do seu pacote. A notação matemática de HMII pode ser expressa por:

  $$\boldsymbol{Heurística\ II}:\ \forall P_i, se\ |P_i| > 1, então\ PMov = DepExterno - DepInterno$$

Em **5** (Figura 1), é verificada a existência de classes que podem ser movimentas entre os pacotes do software. Se não existirem (todas as classes possuem *PMov* <= 0), a estrutura atual é sugerida para o usuário, pois todas as classes estão no pacote visto como o mais adequado por essa abordagem. Caso contrário, a estrutura do software é percorrida, buscando a classe com maior *PMov*. Se duas ou mais classes com o mesmo valor de *PMov* forem encontradas, são utilizadas heurísticas do grupo de decisão, para determinar qual classes deve ser movimentada. As heurísticas desse grupo são utilizadas sequencialmente; se o resultado de uma delas retornar somente uma classe, as demais heurísticas são desconsideradas. O grupo de heurísticas de decisão é composto por:

- Heurística de Decisão I (HDI) - escolher classe com a menor coesão. Essa heurística selecionará para ser movimentada a classe com a menor coesão em seu pacote atual, pois baixa coesão é um indício que a classe está mal posicionada na estrutura do software e que a classe não se assemelha com a função das demais classes do pacote. Isso deteriora a modularização do software, visto que a classe com baixa coesão aumenta a responsabilidade do pacote, pois ela fornece funcionalidade discrepante a do pacote no qual ela está inserida;
- Heurística de Decisão II (HDII) - escolher classe com maior similaridade funcional. Nesse contexto, similaridade funcional é o quanto a funcionalidade oferecida pela classe está relacionada com a funcionalidade proporcionada pelo pacote destino. Portanto, quanto maior a similaridade funcional da classe, mais compatível é a sua função com a função do pacote destino e maior deve ser sua prioridade de ser movimentada. Essa similaridade é calculada pela razão entre a quantidade de classes do pacote destino que se relacionam com a classe a ser movimentada e o total de classes desse pacote;
- Heurística de Decisão III (HDIII) - escolher classe com maior Instabilidade. Nesse contexto, Instabilidade refere-se à medida proposta por Martin [Martin, 1994]. Seguindo a ideia dessa medida, deve-se movimentar a classe com maior Instabilidade, pois essa é mais suscetível a alterações por causa das modificações no software. Sendo assim, ao priorizar o movimento de classes com maior Instabilidade busca-se um software mais estável;

- `Heurística de Decisão IV (HDIV) - escolher classe que gere o menor impacto`. Nesse contexto, impacto é a quantidade de classes que devem ser atualizadas em decorrência de um movimento, para o software não alterar seu funcionamento. Portanto, quanto menor o impacto gerado à estrutura do software, melhor, pois menor quantidade de alterações será necessária para manter o funcionamento do software constante.

Posteriormente, a classe escolhida será movimentada para o seu pacote destino **6** (Figura 1). Por fim, em **7** (Figura 1), as classes que se relacionam com a classe movimentada têm suas dependências atualizadas, para não alterar o comportamento do software. Com a execução desses passos, é obtida a estrutura sugerida, com a qual a SA executa seu processamento, verificando se essa estrutura é melhor que a estrutura original e se deve aceitar o movimento realizado. A vantagem adicionada pela utilização da SA nesse contexto refere-se a evitar mínimos locais. Por isso, utilizando a SA, existe a possibilidade de aceitar um movimento que deteriore a estrutura atual, mas que por consequência acarrete a execução de movimentos que aprimorem a estrutura, obtendo uma estrutura melhor que a anterior. Dessa maneira, o ciclo apresentado na Figura 1 se repetirá até não existirem classes com probabilidade positiva de serem movimentadas. Ao final da reestruturação, a melhor estrutura encontrada é sugerida ao usuário.

5. Resultados e Discussão

Para avaliar a abordagem proposta, 30 sistemas de software pertencentes à categoria de áudio e vídeo foram coletados dos repositórios de código aberto Git (http://git-scm.com/) e Sourceforge (http://sourceforge.net/). Essa categoria agrupa sistemas de software voltados à gravação, à decodificação e à reprodução de mídias de áudio e vídeo. Os sistemas de software coletados estão caracterizados na Tabela 1 e foram escolhidos aleatoriamente da categoria escolhida, mas foram coletados sistemas com variabilidade de tamanho em relação a quantidade de pacotes e classes. Assim, as análises tornam-se mais representativas da totalidade de sistemas existentes no mercado.

Tabela 1 - Caracterização dos Sistemas de Software

#	Software	Quantidade de Pacotes	Quantidade de Classes	#	Software	Quantidade de Pacotes	Quantidade de Classes
S1	Amplitude	12	75	S16	jtag	8	31
S2	ChordAssist	2	34	S17	JTagEditor	13	77
S3	com.sorox.eplug	2	15	S18	JXM	4	27
S4	CoreMP3	9	121	S19	Leipzig	14	44
S5	dubman	2	28	S20	Meloncillo	22	326
S6	FScape	12	178	S21	mp3tagmaster	4	40
S7	Geeboss	38	217	S22	Mr. Random	10	70
S8	jajuk	58	445	S23	musicminer	38	461
S9	JAud	19	112	S24	playlister	4	51
S10	jbuzzer	8	63	S25	podcaster	10	39
S11	JFreedbClient	2	12	S26	PrismiqWeb	3	17
S12	JideoGuard	15	44	S27	soundbot	134	363
S13	jlastfm	3	24	S28	jFLAC	7	63
S14	JOPFilter	5	21	S29	TetraHead	11	148
S15	JSynthLibCVS	70	715	S30	TheJee	11	175

Os sistemas coletados foram submetidos ao processo de reestruturação por meio do *plug-in* desenvolvido para automatizar a abordagem de reestruturação proposta. O tempo médio gasto para realizar as reestruturações foi 2 minutos e 46 segundos, no melhor caso 4 segundos e no pior caso 18 minutos e 32 segundos. Os resultados da reestruturação dos sistemas indicam a porcentagem de melhoria que cada medida e atributo de software apresentou na estrutura sugerida pela reestruturação em relação à estrutura original do software. Por exemplo, para a medida Ca do software `Amplitude` (linha 1 da Tabela

2), supondo o seu valor na estrutura original igual a 10, seu valor na estrutura sugerida é 2,5, pois a melhoria obtida foi de 75%. Analisando os dados da Tabela 2, pode-se afirmar que esses sistemas aprimoraram sua qualidade interna, pois, em todos os casos, as medidas e os atributos analisados tiveram seus valores aprimorados.

Tabela 2 - Porcentagem de Melhoria Proporcionada pela Reestruturação

#	Medidas de Avaliação			Medidas de Reestruturação				Atributo	
	Ca	Ce	$LCOM4_p$	CBO_p	MPC_p	RFC_p	TCC_p	Acoplamento	Coesão
S1	75%	79%	53%	79%	85%	14%	254%	39%	254%
S2	46%	46%	7%	46%	42%	1%	4%	4%	4%
S3	91%	91%	7%	91%	54%	8%	7%	20%	7%
S4	73%	73%	41%	73%	72%	19%	362%	40%	362%
S5	100%	100%	50%	100%	100%	1%	1%	13%	1%
S6	85%	87%	27%	87%	92%	55%	24%	78%	24%
S7	34%	35%	9%	35%	39%	15%	32%	27%	32%
S8	25%	28%	5%	28%	26%	12%	369%	20%	369%
S9	42%	43%	16%	43%	40%	5%	2%	13%	2%
S10	49%	74%	52%	74%	75%	46%	70%	60%	70%
S11	83%	83%	56%	83%	88%	3%	25%	10%	25%
S12	55%	56%	37%	56%	70%	14%	196%	38%	196%
S13	65%	76%	61%	76%	89%	10%	34%	40%	34%
S14	42%	42%	22%	42%	55%	7%	8%	20%	8%
S15	15%	16%	8%	16%	20%	7%	15%	13%	15%
S16	73%	75%	64%	75%	78%	16%	133%	37%	133%
S17	36%	43%	24%	43%	57%	6%	0%	19%	0%
S18	28%	75%	18%	75%	87%	8%	9%	31%	9%
S19	24%	23%	16%	23%	35%	3%	8%	10%	8%
S20	14%	17%	1%	17%	24%	12%	34%	18%	34%
S21	53%	76%	31%	76%	76%	7%	3%	21%	3%
S22	60%	63%	32%	63%	74%	15%	84%	38%	84%
S23	42%	42%	7%	42%	48%	28%	114%	38%	114%
S24	27%	27%	50%	27%	41%	2%	20%	9%	20%
S25	53%	53%	33%	53%	65%	9%	12%	28%	12%
S26	91%	91%	36%	91%	93%	7%	18%	29%	18%
S27	19%	20%	87%	71%	32%	7%	76%	19%	76%
S28	54%	54%	45%	54%	63%	14%	18%	34%	18%
S29	46%	49%	6%	49%	64%	23%	66%	43%	66%
S30	32%	41%	20%	41%	51%	6%	5%	19%	5%

Realizando análise em relação à quantidade de classes movimentadas que a abordagem proposta gerou à estrutura dos sistemas reestruturados (Figura 3), obteve-se que, em média, 26% das classes do software reestruturado foram movimentadas. Isso indica que a abordagem de reestruturação proposta gerou percentual de melhoria ao software (27% - acoplamento e 69% - coesão) maior que o percentual de alterações aplicadas sobre sua estruturada (26% - classes movimentadas). Analisar a movimentação das classes é importante na reestruturação, pois quanto maior essa movimentação, mais modificações são aplicadas ao software e mais esforço do mantenedor será necessário para compreender a estrutura após a reestruturação.

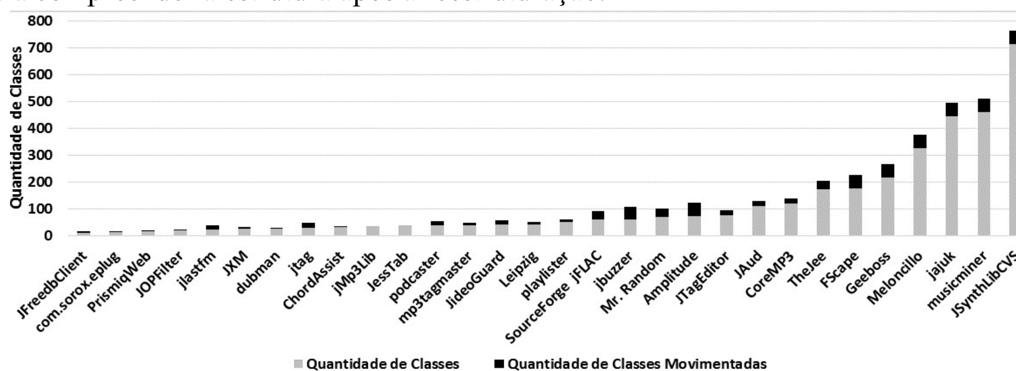

Figura 3 - Relação entre o Total de Classes do Software e a Quantidade de Classes Movimentadas

Analisando a relação entre a quantidade de classes movimentadas e a quantidade de movimentos (precisão dos movimentos realizados), obteve-se 100% de precisão. Isso indica que cada classe foi movimentada uma única vez, sendo remanejada para o "melhor" ponto na estrutura do software, evitando movimentos que não melhorariam sua estrutura. Avaliando se o tamanho dos sistemas reestruturados (quantidade de classes) interfere no ganho proporcionado pela reestruturação (Figura 4), o resultado é negativo. Assim, a abordagem proposta gera ganhos proporcionais ao quanto o software pode ser aprimorado, independentemente do seu tamanho. Portanto, o sistema é aprimorado em relação ao quão deteriorada sua estrutura encontra-se e não ao seu tamanho.

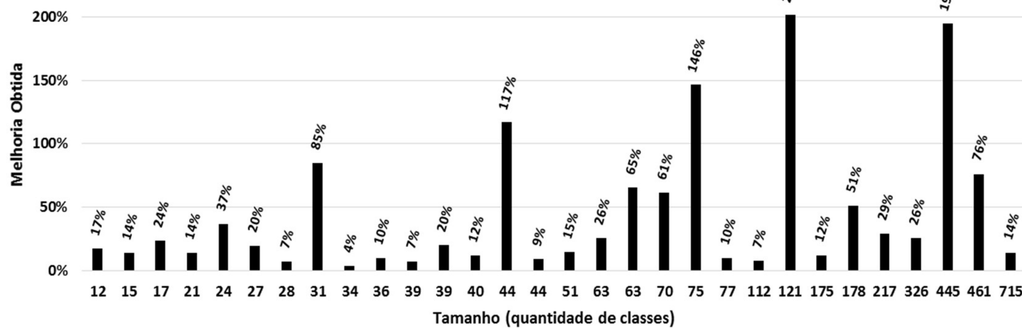

Figura 4 - Relação entre Melhoria Proporcionada e o Tamanho do Software

De forma análoga, foi analisado se o tamanho do sistema reestruturado (quantidade de classes) interfere na quantidade de movimentos realizados pela reestruturação (Figura 5); o resultado é negativo. Sendo assim, a quantidade de classes movimentadas altera-se em função do quanto a estrutura do software está degradada e não ao seu tamanho. Isso ratifica que a abordagem proposta aprimora o software de forma independente do seu tamanho.

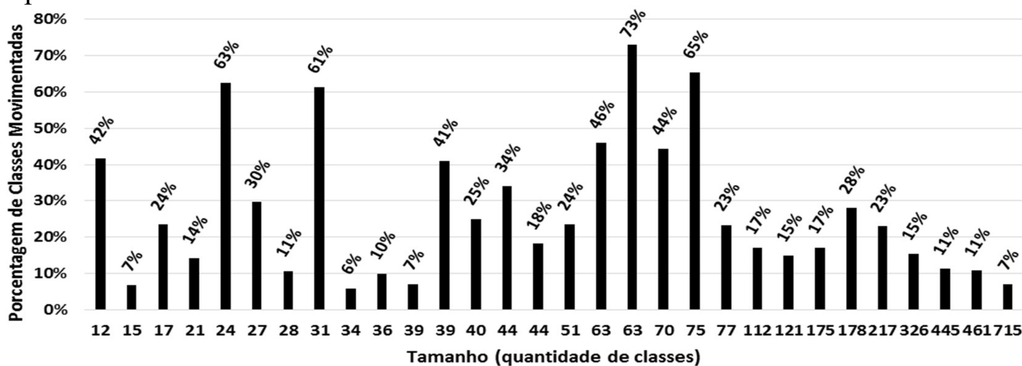

Figura 5 - Relação entre Porcentagem de Classes Movimentadas e Tamanho do Software

Com base nas avaliações apresentadas, percebe-se que a abordagem proposta é capaz de atingir seu objetivo, aprimorando simultaneamente o acoplamento e a coesão do software. Portanto, um software mais modularizado e manutenível é obtido. Além disso, a abordagem proposta é independente do tamanho do sistema reestruturado, aprimorando-o de acordo com o quanto a sua estrutura está degradada.

6. Limitações e Ameaças à Validade

Uma limitação refere-se a não utilização de testes estatísticos robustos para comprovar os resultados apresentados. Uma ameaça que pode influenciar o resultado refere-se à desconsideração da semântica dos pacotes durante a reestruturação, por exemplo, movimentar classes de pacotes da *View* para pacotes da *Controller* em uma arquitetura

MVC (*Model-View-Controller*). Embora o apoio computacional desenvolvido possua funções para respeitar essa semântica, isso somente aplica-se quando o usuário possui conhecimento sobre o sistema para indicar limitações de pacotes que não podem ser reestruturados em conjunto. Esse fato é impraticável nas reestruturações conduzidas, pois os autores não são contribuintes no desenvolvimento do projeto dos sistemas analisados.

Outra ameaça à validade refere-se ao uso de medidas estáticas na abordagem de reestruturação, pois elas podem gerar viesses na análise das dependências e na movimentação de classes entre pacotes. Mesmo que estaticamente exista somente uma dependência entre duas classes, essa dependência pode ser efetivada múltiplas vezes. Por exemplo, uma classe que chama um método de outra classe repetidas vezes dentro de um laço de repetição, aumentando o relacionamento entre tais classes. Portanto, mesmo que estaticamente seja lógico movimentar uma classe entre dois pacotes, em uma análise dinâmica, esse cenário pode ser alterado por causa das múltiplas dependências existentes em tempo de execução. Entretanto, análises dinâmicas sofrem interferências do ambiente [Ernst, 2003]; por isso, nesta investigação, foram utilizadas medidas estáticas.

A quantidade de sistemas utilizados para realizar as análises de eficiência da abordagem proposta pode ser vista como ameaça à validade, pois a amostra pode não ser considerada representativa do total de sistemas existentes. Da mesma forma, a utilização de sistemas de somente uma categoria pode representar outra ameaça, mesmo com a existência de variabilidade de tamanho entre os sistemas utilizados, assemelhando-os ao restante dos sistemas existentes.

7. Lições Aprendidas

Em versões iniciais da abordagem de reestruturação proposta, a medida LCOM foi utilizada dentre as medidas de software que compõem este trabalho. Entretanto, em análises para verificar se essa abordagem atingia o objetivo desejado, foi detectado que, em alguns casos, essa medida não era aprimorada pela de reestruturação, ao contrário das demais medidas utilizadas.

Por causa desse comportamento inesperado, estudos sobre essa medida e sobre os impactos que a reestruturação gerava sobre a LCOM foram aprofundados. Por fim, foi constatado que a medida LCOM não repercute adequadamente os aprimoramentos proporcionados ao software reestruturado. Isso se deve ao fato que, ao movimentar uma classe de um pacote para outro, mesmo que ela possua muitas dependências com o pacote destino, tais dependências podem ser restritas a pequena quantidade de classes em relação ao total de classes desse pacote. Sendo assim, a classe movimentada compartilha dependências com poucas classes do pacote destino e, mesmo melhorando a coesão da classe movimentada e das classes com as quais essa se relaciona, a coesão das demais classes do pacote destino são deterioradas. Isso pode reduzir o valor da medida LCOM do software caso a quantidade de classes existentes no pacote destino seja maior que a quantidade de classes que se relacionam com a classe movimentada.

Para facilitar a compreensão, no exemplo da Figura 6, ao movimentar a `Classe B` do `Pacote1` para o `Pacote2`, sua coesão e a da `Classe C` aumentarão, pois elas possuem dependências em comum. Porém, a coesão da `Classe D`, da `Classe E` e da `Classe F` reduzirão (aumento do valor da medida LCOM), pois elas não possuem dependências em comum com a `Classe B`. Com essa lição, espera-se que futuros trabalhos que abordem reestruturação de software não trilhem o mesmo caminho, evitando abordagens infrutíferas e perda de recursos.

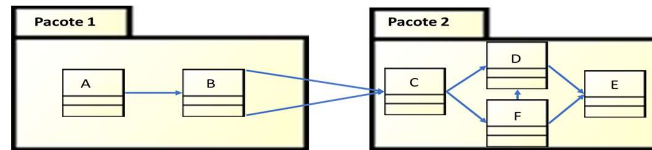

Figura 6 - Exemplo de Deterioração com LCOM

8. Considerações Finais

Neste trabalho, foi proposta uma abordagem para reestruturação de software baseada na movimentação de classes entre pacotes. Essa reestruturação age na contracorrente da degradação que os sistemas de software sofrem ao longo do seu ciclo de vida, tornando-os menos acoplados e mais coesos. Para avaliar essa abordagem, 30 sistemas de software Java foram submetidos ao processo de reestruturação, coletando dados antes e após a reestruturação. Com base nos dados obtidos, pode-se afirmar que a abordagem proposta atinge seu objetivo de tornar os sistemas menos acoplados e mais coesos, pois, em todos os casos, a reestruturação diminuiu o acoplamento e aumentou a coesão.

Além disso, em todos os casos, a reestruturação proposta aprimorou o valor das medidas utilizadas, preservando a estrutura original de pacotes existente no software, sem acrescentar ou remover pacotes, independentemente do tamanho do sistema reestruturado. Outro ponto positivo da abordagem de reestruturação proposta é o baixo impacto gerado à estrutura do software reestruturado, movimentando em média apenas 26% de suas classes e levando em média 2 minutos e 46 segundos para realizar a reestruturação.

Como trabalhos futuros, pretende-se: i) conduzir estudo com maior quantidade de sistemas de software; ii) utilizar sistemas pertencentes a diferentes categorias de software; iii) aplicar estatísticas mais robustas para ratificar a melhoria obtida; iv) investigar possibilidades para aumentar ganhos proporcionados aos atributos e medidas analisadas; v) realizar comparações da abordagem proposta com os trabalhos relacionados; e vi) considerar medidas dinâmicas para determinar a movimentação de classes.

Referências

Arnold, R. S. Software Restructuring. In: IEEE. pp. 607-617. 1989.

Barbosa, N.; Hirama, K. Assessment of Software Maintainability Evolution Using C&K Metrics. In: IEEE Latin America Transactions. pp. 1232-1237. 2013.

Bhatt, P.; Shroff, G.; Misra, A. K. Dynamics of Software Maintenance. In: SIGSOFT Software Engineering Notes, pp. 1-5. 2004.

Bianchi, A.; Caivano, D.; Lanubile, F.; Visaggio, G. Evaluating Software Degradation through Entropy. In: International Software Metrics Symposium. pp. 210-219. 2001.

Bieman, J. M.; Kang, B. K. Cohesion and Reuse in an Object-Oriented System. In: Software Engineering Notes ACM. pp. 259-262, 1995.

Bryton, S.; Abreu, F. B. Modularity-Oriented Refactoring. In: Software Maintenance and Reengineering. pp. 294-297. 2008.

Chidamber, S. R.; Kemerer, C. F. A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering. pp. 476-493. 1994.

Du Bois, B.; Demeyer, S.; Verelst, J. Refactoring-Improving Coupling and Cohesion of Existing Code. In: Reverse Engineering. pp. 144-151. 2004.

Erdil, K.; Finn, E.; Keating, K.; Meattle, J.; Park, S. Software Maintenance as Part of the Software Life Cycle. In: Comp180: Software Engineering Project. pp. 1-49. 2003.

Erlikh, L. Leveraging Legacy System Dollars for e-Business. In: IT professional. pp. 17-23. 2000.

Ernst, M. D. Static and Dynamic Analysis: Synergy and Duality. In: International Conference on Software Engineering Workshop on Dynamic Analysis. pp. 24-27. 2003.

Focus, M. Measures and Metrics. Disponível em: <http://supportline.microfocus.com/documentation/books/ev56/ev56books/acrobat/Measures%20and%20Metrics.PDF>. Acessado em: Março de 2016.

Gurp, V. J.; Bosch, J. Design Erosion: Problems and Causes. In: Journal of Systems and Software. pp. 105-119. 2002.

Gyimothy, T. To Use or Not to Use? The Metrics to Measure Software Quality (Developers' View). In: European Conference on Software Maintenance and Reengineering. pp. 3-4. 2009.

Hitz, M.; Montazeri, B. Measuring Coupling and Cohesion in Object-Oriented Systems. In: International Symposium on Applied Corporate Computing. pp. 1-10. 1995.

Honglei, T.; Wei, S.; Yanan, Z. The Research on Software Metrics and Software Complexity Metrics. In: Computer Science-Technology and Applications. pp. 131-136. 2009.

ISO/IEC 25000. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation. 2014.

Kan, S. H. Metrics and Models in Software Quality Engineering. In: Addison-Wesley.560p. 2002.

Kirkpatrick, S. Optimization by simulated annealing: Quantitative studies. Journal of Statistical Physics pp. 671-680. 1983.

Lanza, M.; Marinescu, R. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Orientated Systems. 206p. 2006.

Lee Y.; Liang B.; Wu S.; Wang F. Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow. In: International Conference on Software Quality. pp. 81-90. 1995.

Lehman, M. Program, Life-Cycle, and the Law of Program Evolution. In: IEEE. pp. 1060-1076. 1980.

Lima, E. D. C.; Resende A. M. P. Uma Análise dos Valores de Referência de Algumas Medidas de Software. 192p. Dissertação. Universidade Federal de Lavras. Brasil. 2014.

Martin, R. OO design quality metrics. An analysis of dependencies. In: Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics. pp. 151-170. 1994.

Opdyke, W.F. Refactoring Object-Oriented Frameworks. 206p. Dissertação. University of Illinois.1992.

Palomba, F.; Tufano, M.; Bavota, G.; Oliveto, R.; Marcus, A.; Poshyvanyk, D.; De Lucia, A. Extract Package Refactoring in ARIES. In: International Conference on Software Engineering. pp. 669-672. 2015.

Pinto, F.; Costa, H. Melhoria da Qualidade da Estrutura Interna de Sistemas de Software por Redução do Nível de Acoplamento entre Pacotes. In: Simpósio Brasileiro de Qualidade de Software. pp. 194-208. 2014.

Pressman, R.; Maxim, B. Software Engineering: A Practitioner's Approach. McGraw-Hill. 976p. 2014.

Santos, D. B.; Resende, A.; Junior, P. A., Costa, H. Attributes and Metrics of Internal Quality that Impact the External Quality of Object-Oriented Software: A Systematic Literature Review. In: Latin American Computing Conference. 2016.

Silva, R.; Costa, H. Graphical and Statistical Analysis of the Software Evolution Using Coupling and Cohesion Metrics - An Exploratory Study. In: Latin American Computing Conference. pp. 1-9. 2015.

Sneed, H.M.; Brössler, P. Critical Success Factors in Software Maintenance: A Case Study. In: International Conference on Software Maintenance. pp. 190-198. 2003.

Sommerville, I. Software Engineering. Addison-Wesley. 792p. 2010.

Swanson, E. B. IS "Maintainability": Should it Reduce the Maintenance Effort?. ACM SIGMIS. pp. 65-76. 1999.

Zanetti, M. S.; Tessone, C. J.; Scholtes, I.; Schweitzer, F. Automated Software Remodularization Based on Move Refactoring. In: International Conference on Modularity. pp. 73-84. 2014.