DANILO BATISTA DOS SANTOS

UMA ABORDAGEM PARA REESTRUTURAÇÃO DE SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS UTILIZANDO MEDIDAS DE ACOPLAMENTO E COESÃO

LAVRAS-MG
2017

DANILO BATISTA DOS SANTOS

UMA ABORDAGEM PARA REESTRUTURAÇÃO DE SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS UTILIZANDO MEDIDAS DE ACOPLAMENTO E COESÃO

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software e Banco de dados, para a obtenção do título de Mestre.

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2017

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Santos, Danilo Batista dos.
Uma Abordagem Para Reestruturação De Sistemas De Software Orientados A Objetos Utilizando Medidas De Acoplamento E Coesão / Danilo Batista dos Santos. - 2017.
132 p. : il.

Orientador(a): Heitor Augustus Xavier Costa.

Dissertação (mestrado acadêmico) - Universidade Federal de Lavras, 2017.
Bibliografia.

1. Engenharia de Software. 2. Reestruturação. 3. Medidas de Acoplamento e Coesão. I. Costa, Heitor Augustus Xavier. II. Título.

Danilo Batista dos Santos

UMA ABORDAGEM PARA REESTRUTURAÇÃO DE SISTEMAS DE SOFTWARE ORIENTADOS A OBJETOS UTILIZANDO MEDIDAS DE ACOPLAMENTO E COESÃO

AN APPROACH TO RESTRUCTURE OBJECT ORIENTED SOFTWARE USING SOFTWARE METRICS OF COUPLING AND COHESION

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software e Banco de dados, para a obtenção do título de Mestre.

APROVADA em 27 de janeiro de 2017.
Dr. Antônio Maria Pereira de Resende UFLA
Dr. Paulo Afonso Parreira Junior UFLA
Dr. Matheus Carvalho Viana UFSJ
Dr. Diego Roberto Colombo Dias UFSJ

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS-MG
2017

AGRADECIMENTOS

Agradeço a **DEUS** por me dar o dom da vida, por ser bom o tempo todo, por me dar força a cada segundo para transformar as pedras postas no meu caminho em uma escada para alcançar essa vitória e por todas as demais bênçãos.

Agradeço aos meus pais Lázaro e Aparecida por me ensinarem a riqueza da simplicidade e a necessidade do trabalho, pelo amparo nas condições mais adversas e por sempre primarem pela educação de seus filhos, não poupando esforços e o suor para vê-los vencer. Eu me orgulho dos meus pais.

Agradeço as minhas amadas irmãs Vanilha Santos e Dalilhia Santos e ao meu "irmão" Eder Diniz pelos sábios conselhos, atenção, companheirismo, apoio, orientação e momentos de alegria e distração que tornaram essa caminhada mais leve.

Agradeço a minha amada namorada Ingrid Morais por ser minha amiga e fiel companheira, por me ajudar em cada obstáculo, ouvindo meus desabafos e me aconselhando. Obrigado por alegrar cada dia da minha vida com seu sorriso.

Agradeço ao meu orientador Heitor Costa pelos ensinamentos e empenho dedicado ao nosso projeto de pesquisa. Sem seu ideal inicial, orientação, conhecimento e experiência, o objetivo e os bons frutos colhidos não existiriam.

Agradeço aos professores do PPGCC/UFLA pelos ensinamentos, pelo conhecimento compartilhado, pelas discussões enriquecedoras e pelos desafios impostos que me fizeram crescer não somente academicamente, mas também como pessoa.

Agradeço aos meus amigos do PPGCC/UFLA pelas horas de estudo, apoio, alegrias, tristezas e experiências compartilhadas ao longo desses dois anos. Agradeço também aos amigos de vida que me incentivaram e deram força e alegria para que esse objetivo fosse alcançado.

Agradeço também a algumas pessoas especiais em minha vida que, cada uma de sua forma, me ajudaram nessa vitória. Yuri Diniz, Antônio de Oliveira, Romilda Moraes, Cristiano Garcia, Wagner Nascimento, Daniel Ferreira e ao ap. 210 (Brejão).

Agradeço à Universidade Federal de Lavras pela oportunidade, infraestrutura, financiamento e demais recursos concedidos para a realização deste mestrado. Agradeço também aos funcionários e servidores do Departamento de Ciência da Computação da UFLA, pelo apoio nesta conquista.

Essa vitória não é só minha, mas de todos aqueles que torceram contra ou a favor dela.

RESUMO

Desde as fases iniciais do desenvolvimento do sistema de software, uma das principais preocupações é com a qualidade desses sistemas. Para alcançar esse objetivo, esses sistemas devem evoluir constantemente para atender às necessidades dos usuários e do ambiente em que está inserido - *lei da mudança contínua*. Por isso, são conduzidas manutenções para corrigir, adicionar ou adaptar funções no sistema. Entretanto, se essas manutenções forem conduzidas em discordância com os padrões de projeto empregados e às melhores práticas de programação, o sistema se tornará progressivamente mais acoplado e menos coeso, sendo menos modularizado e degradando a sua qualidade interna. Para agir na contracorrente dessa degradação, neste trabalho, é apresentada uma Abordagem para Reestruturar Sistemas de Software (ARS), por meio da movimentação de classes entre pacotes, para gerar sistemas com melhor qualidade interna, mais coesos e menos acoplados, tornando-os mais modulares e manuteníveis. Para alcançar esse objetivo, ARS apresenta um conjunto de passos bem definidos para realizar a reestruturação, utilizando medidas de software para determinar o estado do sistema e heurísticas desenvolvidas para tomar decisões ao longo da reestruturação. ARS também é apoiada pela heurística *Simulated Annealing*, para evitar a subjetividade humana e aumentar as chances de sucesso da sua aplicação. O resultado da avaliação de ARS revela sua capacidade de aprimorar a qualidade estrutural de sistemas de software, pois essa abordagem é capaz de melhorar os valores das sete medidas de software utilizadas de forma estatisticamente significativa e reduzir a quantidade de dependências entre seus pacotes, gerando sistemas mais modulares e manuteníveis. Em outra avaliação, ARS foi comparada com outras abordagens existentes, indicando que ARS é mais eficiente do que elas. Esse resultado foi obtido por meio de um teste multivariado de médias, que comprovou que as estruturas sugeridas por ARS apresentaram maior ganho aos valores das medidas de software utilizadas e, consequentemente, melhor estrutura interna que as demais abordagens. Portanto, a aplicação de ARS é capaz de aprimorar a qualidade estrutural do sistema, agindo na contracorrente da degradação da qualidade ocasionada pelas manutenções mal planejadas. Desse modo, ARS possibilita que futuras manutenções sejam realizadas com menos dificuldade, reduzindo os recursos e esforços empregados na sua condução.

Palavras-chave: Reestruturação de Software. Qualidade de Software. Medidas de Software.

ABSTRACT

Since the early stages of software development, one of the main concerns is that it has quality. To achieve this goal, software must constantly evolve to meet the needs of users and the environment in which it is embedded - law of continuous change. Therefore, maintenance is conducted to correct, add or adapt features in the software. However, if such maintenance is conducted in disagreement with the design standards employed and best programming practices, the software will become progressively more coupled and less cohesive, becoming less modularized, and degrading its internal quality. In order to act in the countercurrent of this degradation, this work presents an approach to restructuring software (ARS) through the movement of classes between packages, to generate software with better internal quality, more cohesive and less coupled, making them more modular and maintainable. To achieve this goal, the ARS presents a set of well-defined steps to carry out the restructuring process. In this process software measures are used to determine the state of the software and heuristics to make decisions throughout the restructuring. This approach is also supported by the Simulated Annealing heuristic, to avoid human subjectivity and increase the chances of successful application of ARS. The result of the evaluation of ARS reveals its ability to improve the structural quality of the software. This improvement was proven, since this approach is able to improve the values of the seven measures of software used in a statistically significant way and to reduce the amount of dependencies between its packages, generating more modular and maintainable software. In another evaluation, ARS was compared to other approaches, indicating that ARS is more efficient than them. This result was obtained through a multivariate test of means, which proved that the structures suggested by ARS presented greater gain to the values of the software measures used and, consequently, a better internal structure than the other approaches. Therefore, the application of ARS is able to improve the structural quality of the software, acting in the countercurrent of the degradation of its quality caused by the badly planned maintenance. In this way, ARS makes possible for future maintenance to be carried out with less difficulty, reducing the resources and efforts used to drive it.

Keywords: Software Restructuring. Software Quality. Software Measures.

LISTA DE FIGURAS

Figura 2.1 - Método da Pesquisa 22

Figura 2.2 - Mapa Conceitual da Abordagem Proposta para Reestruturação de Sistemas de Software 23

Figura 4.1 - Diagrama de Sequência do Exemplo para as Medidas MPC, CBO e RFC . 34

Figura 4.2 - Diagrama de Classes do Exemplo para as Medidas MPC, CBO e RFC 34

Figura 4.3 - Exemplo para as Medidas Ca e Ce 36

Figura 4.4 - Exemplo para a Medida LCOM4 38

Figura 4.5 - Grafo Gerado pelo Exemplo para a Medida LCOM4 38

Figura 4.6 - Diagrama de Classes para o Exemplo para a Medida TCC 39

Figura 4.7 - Diagrama de Sequência para o Exemplo para a Medida TCC 39

Figura 5.1 - Pseudocódigo *Simulated Annealing* 43

Figura 5.2 - Mínimos Locais e Mínimo Global no Espaço de Busca 45

Figura 6.1- Exemplo da Justificativa de Alteração na Granularidade das Medidas: (a) Antes do Movimento e (b) Depois do Movimento 47

Figura 6.2 - Visão Geral de ARS 60

Figura 6.3 - Diagrama de Pacotes do Sistema de Software Exemplo 63

Figura 6.4 - Diagrama de Sequência do Sistema de Software Exemplo 63

Figura 6.5 - Diagrama de Pacotes do Sistema de Software Após Movimentar a Classe Classe5 do Pacote Pacote5 para o Pacote Pacote2 69

Figura 7.1 - *Java elements* Presentes no *Workspace* 74

Figura 7.2 - Mapeamento do Código para AST 74

Figura 7.3 - Projeto Arquitetural do Apoio Computacional (*Plug-in*) 75

Figura 7.4 - Tipo Abstrato de Dados para Armazenar Sistemas de Software em SRT ... 77

Figura 7.5 - Ícones de SRT na Barra de Ícones do Eclipse 78

Figura 7.6 - Tela de Configuração dos Limites Superiores de Deterioração 78

Figura 7.7 - Tela de Execução da Reestruturação 79

Figura 7.8 - Recorte da Tela para Visualização do Diagrama de Dependências Interclasses 80

Figura 7.9 - Recorte da Tela de Restrição de Pacotes Aba *Restrict Packages* 81

Figura 7.10 - Recorte da Tela de Log de Movimentação de Classes - Aba *View Moviment Log* 82

Figura 8.1 - Classificação da Distribuição de Frequência de Acordo com a Curtose 96

Figura 8.2 - Resultado Teste *T2 de Hotelling* 98

Figura 8.3 - Relação Entre a Porcentagem de Classes Movimentadas e o Tamanho dos Sistemas de Software 101

Figura 8.4 - Quantidade de Dependências Entre as Estruturas Original e Sugerida 102

Figura 8.5 - Condução da Avaliação Externa 104

Figura 8.6 - Resultado do Teste de Normalidade dos Dados da Avaliação Externa 111

Figura 8.7 - Resultado do Teste T2 de Hotelling sobre os Dados da Avaliação Externa 112

Figura 9.1 - Exemplo de Deterioração com a Medida LCOM 120

LISTA DE TABELAS

Tabela 3.1 - Sumarização dos Trabalhos Relacionados 29
Tabela 6.1 - Cálculo da Medida MPC_p 64
Tabela 6.2 - Cálculo da Medida CBO_p 64
Tabela 6.3 - Cálculo da Medida RFC_p 64
Tabela 6.4 - Cálculo da Medida Ca 65
Tabela 6.5 - Cálculo da Medida Ce 65
Tabela 6.6 - Cálculo da Medida $LCOM_p$ 66
Tabela 6.7 - Cálculo da Medida TCCp 66
Tabela 6.8 - Valores das Medidas para o Sistema de Software 67
Tabela 6.9 - Probabilidade de Movimentação das Classes do Sistema de Software 68
Tabela 6.10 - Valores Atualizados das Medidas para as Classes do Sistema de Software 69
Tabela 6.11 - Valores Atualizados de LCOM4p para os Pacotes 69
Tabela 6.12 - Valores Atualizados das Medidas Utilizadas para o Sistema de Software 70
Tabela 8.1 - Caracterização dos Sistemas de Software Selecionados 84
Tabela 8.2 - Trabalhos Relacionados 85
Tabela 8.3 - Valor das Medidas dos Estados Original e Sugerido dos Sistemas de Software e Melhoria Obtida nas Medidas de Software Utilizadas 90
Tabela 8.4 - Estatística Descritiva 94
Tabela 8.5 - Intervalos de Confiança Simultâneos 99
Tabela 8.6 - Dados do Processo de Reestruturação 100
Tabela 8.7 - Sistemas de Software Utilizados na Avaliação Externa 105
Tabela 8.8 - Estados Original e Sugerido por SRT e por ARIES e Diferença entre esses Estados dos Sistemas de Software Analisados 107
Tabela 8.9 - Estatística Descritiva da Avaliação Externa 110
Tabela 8.10 - Intervalos de Confiança dos Dados da Avaliação Externa 113

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Justificativa 15
1.2 Objetivo 16
1.3 Estrutura do trabalho 17

2 DESENVOLVIMENTO DA PESQUISA 18
2.1 Considerações iniciais 18
2.2 Tipo de pesquisa 18
2.3 Método de pesquisa 20
2.4 Considerações finais 25

3 TRABALHOS RELACIONADOS 26

4 MEDIDAS DE SOFTWARE 30
4.1 Considerações iniciais 30
4.2 Métricas, medidas e medição 30
4.3 Classificação das medidas 31
4.4 Medidas de acoplamento 32
4.5 Medidas de coesão 37
4.6 Considerações finais 39

5 *SIMULATED ANNEALING* 41
5.1 Considerações iniciais 41
5.2 A heurística 42
5.3 O pseudocódigo 42
5.4 Considerações finais 45

6 ARS - ABORDAGEM PARA REESTRUTURAÇÃO DE SOFTWARE 46
6.1 Considerações iniciais 46
6.2 Adequação das medidas de software 46
***6.3* Adaptação da s*imulated annealing* 51**
6.4 Definição das heurísticas 54
6.4.1 Heurísticas de movimentação 54
6.4.2 Heurísticas de decisão 55
6.4.3 Heurísticas de convergência 58
6.4.4 Descrição da abordagem para reestruturação de sistemas de software 59
6.5 Exemplo de aplicação da abordagem para reestruturação de sistemas de software 62
6.6 Considerações finais 71

7 SRT - *SOFTWARE RESTRUCTURING TOOL* 72
7.1 Considerações iniciais 72
7.2 Tecnologias empregadas 73
7.3 Arquitetura de SRT 74
7.4 Estrutura de dados do software de SRT 76
7.5 Exemplo de utilização de SRT 77
7.6 Considerações finais 82

8 AVALIAÇÃO DA ABORDAGEM 83
8.1 Considerações iniciais 83
8.2 Caracterização dos sistemas de software utilizados 83
8.3 Avaliação interna 87
8.4 Avaliação externa 102
8.5 Discussão dos resultados 113
8.5.1 Resultados da avaliação interna 113
8.5.2 Resultados da avaliação externa 115
8.6 Considerações finais 116

9 CONSIDERAÇOES FINAIS 118
9.1 Conclusões 118
9.2 Contribuições 119
9.3 Lições aprendidas 120
9.4 Ameaças à validade 121
9.5 Limitações 122
9.6 Trabalhos futuros 123
9.7 Resultados de publicações 124

REFERÊNCIAS 125

1 INTRODUÇÃO

Com o aumento da competividade no mercado de sistema de software, a necessidade de entregar produtos de qualidade ganha mais força a cada dia (JOHN; KADADEVARAMATH; IMMANUEL, 2016). Por isso, continuamente, os engenheiros de software esforçam-se para agregar qualidade a esses sistemas, propondo soluções de projeto adequadas ou identificando pontos que possam ter sua qualidade aprimorada (AL DALLAL, 2013). Essa qualidade representa o grau de capacidade desses sistemas satisfazerem às necessidades explícitas e implícitas das partes envolvidas no projeto (*stakeholders*) (ISO/IEC 25010, 2011; SWEBOK, 2014; PRESSMAN; MAXIM, 2016). A busca em prover e manter essa qualidade é a base dos esforços da Engenharia de Software (SWEBOK, 2014), a qual tem à disposição normas, modelos, metodologias e padrões de qualidade para apoiar nessa busca.

Entretanto, não basta somente proporcionar qualidade aos sistemas de software durante seu desenvolvimento, deve-se preocupar com a manutenção dessa qualidade ao longo do tempo, pois, naturalmente, a estrutura desses sistemas tende a deteriorar (conforme a **lei da mudança contínua** (LEHMAN, 1980)). De acordo com essa lei, a qualidade de um sistema de software diminui a menos que ele seja adaptado para refletir mudanças ocorridas em seu ambiente. Portanto, tal lei explicita as constantes e indispensáveis manutenções aplicadas no ciclo de vida desses sistemas. Com isso, sistemas de software têm sua vida útil prolongada, pois, se eles têm qualidade, tendem a atender melhor às necessidades dos usuários.

Por isso, os sistemas de software estão constantemente sob o processo de manutenção, que se refere à ação de modificá-los 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) (MAMONE, 1994; ERDIL et al., 2003; PFLEEGER; ATLEE, 2009; SOMMERVILLE, 2015; PRESSMAN; MAXIM, 2016). Contudo, independentemente do tipo da manutenção, esse processo é o mais oneroso no ciclo de vida do software, consumindo entre 70% e 90% dos seus recursos (ERDIL et al., 2003; ERLIKH, 2000, PRESSMAN; MAXIM, 2016). Além disso, esse é o processo que exige mais esforço dentre as atividades relacionadas à Engenharia de Software (SNEED; BRÖSSLER, 2003). Essas afirmações abrangem sistemas de software proprietários e de código aberto (DAFFARA et al., 2000). Mesmo que não se possa fazer alusão direta a termos financeiros em sistemas de código aberto, tem-se o tempo gasto em horas/homem e outros recursos pessoais (SCACCHI, 2007), que podem ser compreendidos como recursos consumidos para a manutenção.

Todavia, ainda que o processo de manutenção seja oneroso, sua execução é essencial, pois fornece "sobrevida" ao sistema de software ao aprimorar sua qualidade, assegurando mais competitividade e qualidade ao produto (sistema) final oferecido ao cliente (ISO/IEC 25000, 2014; PADUELLI, SANCHES, 2006). Assim, conforme a **lei da mudança contínua**, sistemas de software devem evoluir constantemente para não se tornarem progressivamente menos satisfatório ao usuário (LEHMAN, 1980; PRESSMAN; MAXIM, 2016). Essas manutenções devem ser planejadas e conduzidas de forma apropriada para não reduzir a modularidade e a manutenibilidade desses sistemas e, consequentemente, a sua qualidade. A modularidade é uma subcaracterística de manutenibilidade, que se refere à capacidade do sistema de software de ser modificado de forma efetiva e eficiente (ISO/IEC 25010, 2011). Por isso, recomenda-se que esses sistemas possuam alto grau de manutenibilidade (ISO/IEC 25000, 2014) para não demandar gastos excessivos de tempo e de recursos e para manter/aumentar a qualidade deles na execução de futuras manutenções.

Com o objetivo de avaliar essa qualidade ao longo das manutenções, podem ser utilizadas medidas de software (HONGLEI; WEI; YANAN, 2009). Essas medidas, variáveis cujo valor é associado como resultado do processo de medição (ISO/IEC 25000, 2014), mensuram atributos específicos de sistemas de software (*e.g.*, acoplamento e coesão). A utilização dessas medidas fornece dados quantitativos (LI; HENRY, 1993; HONGLEI; WEI; YANAN, 2009), para dar suporte a tomada de decisão dos engenheiros de software (GYIMOTHY, 2009) no processo de aprimoramento da qualidade desses sistemas. Além disso, medidas de software podem indicar entidades do software (*e.g.* métodos, classes e pacotes) que apresentam valores "ruins" de acordo com determinada medida. Dessa maneira, essas entidades podem ser reestruturadas, movimentando-as para outro ponto na estrutura do sistema de software para aprimorar a medida analisada (LANZA; MARINESCU, 2006). Por exemplo, em um sistema de software hipotético com 50 classes, seja uma classe `C` com a medida `X` igual a `100` e as demais classes apresentam valor máximo igual a `10` para essa medida; logo, uma análise minuciosa sobre essa classe (com valor discrepante para a medida `X`) deve ser realizada, pois pode representar uma possibilidade de reestruturação.

Nesse ponto, observa-se que as medidas de software apresentam estreito relacionamento com a reestruturação, que se refere ao processo de realizar alterações arquiteturais no sistema de software para melhorar sua qualidade interna, sem interferir no seu comportamento perante o usuário (ARNOLD, 1989; OPDYKE, 1992). Esse relacionamento deve-se ao fato que as medidas indicam pontos na estrutura do sistema de software que apresentam baixa qualidade

em relação a uma medida analisada. Logo, a reestruturação pode ser aplicada nesses pontos para aprimorar a medida analisada e gerar ganhos à qualidade estrutural desse sistema.

Além disso, com a utilização de medidas de software, são encontradas oportunidades para realizar reestruturação (DU BOIS; DEMEYER; VERELST, 2004). Dessa maneira, percebe-se que, realizando a reestruturação com o auxílio de medidas de software, obtém-se uma forma automática de aprimorar a qualidade interna de sistemas de software, melhorando sua modularidade e sua manutenibilidade, sem o auxílio de um ser humano para indicar oportunidades de melhoria e guiar a reestruturação. Esse fato pode auxiliar engenheiros de software na busca por oportunidades de aperfeiçoamento desses sistemas, facilitando a execução dessa atividade (ABDEEN et al., 2013).

1.1 Justificativa

Para manter a qualidade do sistema de software, não basta simplesmente executar o processo de manutenção. Esse processo deve ser executado empregando as melhores práticas (e.g. sem duplicidade, bem organizado) de desenvolvimento e padrões de projeto para obter resultados de qualidade (TERRA; VALENTE, 2010). Caso contrário, gera-se degradação na qualidade do software (BIANCHI et al., 2001; LEHMAN, 1980), também denominada erosão de software (PARNAS, 1994; GURP; BOSCH, 2002) ou envelhecimento de software, levando à redução da modularidade e da manutenibilidade do sistema de software (BIANCHI et al., 2001) e ao aumento do custo e do risco de manutenções futuras.

Ainda que a primeira versão do sistema de software seja desenvolvida utilizando as melhores práticas e em conformidade com normas e modelos de qualidade, satisfazendo as necessidades dos seus clientes desde o início de seu ciclo de vida, sua qualidade tende a deteriorar por causa das sucessivas manutenções (STEPHEN et al., 2001). Isso ocorre, pois, as manutenções não são planejadas e não são estruturadas, visando apenas à correção de erros ou à inserção de funções “às pressas”, sem importar-se com fatores de qualidade, padrões de projeto ou estrutura existente no sistema de software, gerando um sistema de software com código desorganizado, ilegível e propenso a falhas (ERDIL et al., 2003).

Além disso, o mantenedor pode agravar a degradação ao inserir entidades em pontos inadequados da estrutura do sistema de software aumentando o acoplamento[1] e reduzindo a coesão[2] (atributos indicadores da qualidade interna) dos sistemas de software (LANZA;

[1] Grau de dependência entre duas entidades.
[2] Grau em que os componentes de uma entidade estão relacionados.

MARINESCU, 2006; CHEN et al., 2002; BAVOTA; MARCUS; OLIVETO, 2013; AL DALLAL, 2013), prejudicando a modularidade e a manutenibilidade desses sistemas (GURP; BOSCH, 2002). Em situações em que sucessivas manutenções dessa forma são executadas, sistemas de software tornam-se progressivamente menos modulares e manuteníveis, gerando sua substituição no mercado ou expressiva necessidade de redesenvolvimento (SARKAR et al., 2009).

Entretanto, reprojetar totalmente um sistema de software demanda significativa quantidade de recursos (ESPINDOLA; MAJDENBAUM; AUDY, 2004). Portanto, agir na contracorrente dessa degradação é importante para evitar os problemas por ela ocasionados e gastos excessivos na recuperação do sistema de software. Uma forma de combater essa degradação é por meio da reestruturação (ARNOLD, 1989; FOWLER, 1999; SILVA; BALASUBRAMANIAM, 2012), tais como, as empregadas nos trabalhos relacionados desta investigação:

a) Movimentar classes entre pacotes (ABDEEN et al., 2009; ZANETTI et al., 2014; PINTO; COSTA, 2014);
b) Decompor pacotes em entidades menores (BAVOTA; MARCUS; OLIVETO, 2010; BAVOTA; MARCUS; OLIVETO, 2013; (PALOMBA et al., 2015);
c) Unir/dividir pacotes (ABDEEN et al., 2013).

Essa reestruturação visa recuperar/melhorar a qualidade existente no sistema de software, melhorando sua modularidade. Essa melhoria incorre em aumentar a manutenibilidade, permitindo realizar alterações nesse sistema com menos dificuldade, levando à redução dos recursos gastos durante a condução de futuras manutenções (SOMMEVILLE, 2010). Portanto, reestruturação criteriosa e bem-feita pode gerar ganhos financeiros e melhorar a qualidade de sistemas de software. Sendo assim, a proposição de uma abordagem de reestruturação torna-se essencial para diminuir a degradação da qualidade ocasionada pelas manutenções mal planejadas, as quais geram perdas financeiras e perdas na qualidade interna (redução da modularidade e da manutenibilidade) desses sistemas (BAVOTA; MARCUS; OLIVETO, 2013).

1.2 Objetivo

O objetivo foi aperfeiçoar a qualidade interna de sistemas de software por meio de uma abordagem de reestruturação baseada na movimentação de classes entre pacotes. Desse modo, espera-se alcançar baixo acoplamento e alta coesão dos pacotes e, consequentemente, melhorar

a modularidade e a manutenibilidade de sistemas de software. Para atingir esse objetivo, os seguintes objetivos específicos foram atendidos:

a) Identificar estado da arte em relação a Qualidade de Software, Manutenção de Software, Reestruturação de Software e Medidas de Software;
b) Escolher medidas de acoplamento e de coesão para integrarem a abordagem de reestruturação;
c) Estudar e avaliar as técnicas/heurísticas de Inteligência Artificial para serem utilizadas na solução do problema;
d) Construir e otimizar uma abordagem de reestruturação de sistemas de software;
e) Desenvolver, testar e otimizar um *plug-in* para a plataforma Eclipse IDE, para automatizar a abordagem desenvolvida e possibilitar avaliá-la;
f) Coletar sistemas de software para avaliar a abordagem;
g) Realizar a avaliação da abordagem elaborada para verificar sua capacidade de aprimorar a qualidade interna de sistemas de software.

1.3 Estrutura do trabalho

O presente trabalho encontra-se estruturado em 9 capítulos. No Capítulo 2, é apresentada a descrição das fases e das atividades da metodologia de desenvolvimento desta investigação. No Capítulo 3, são descritos alguns trabalhos relacionados. No Capítulo 4, são descritas e exemplificadas as medidas de software utilizadas na abordagem para reestruturação de sistemas de software. No Capítulo 5, são tratados aspectos gerais da heurística de Inteligência Artificial utilizada no desenvolvimento deste trabalho (*Simulated Annealing*). No Capítulo 6, são apresentados uma abordagem de reestruturação de sistemas de software, os detalhes de funcionamento e um exemplo de uso. No Capítulo 7, são abordados a ferramenta desenvolvida para automatizar a abordagem e um exemplo prático de sua utilização. No Capítulo 8, é apresentada e discutida a avaliação de eficiência da abordagem, para verificar se a estrutura sugerida é melhor que a estrutura original do sistema software e para compará-la com abordagens semelhantes. No Capítulo 9, são exibidas conclusões, contribuições, lições aprendidas, ameaças à validade e limitações desta investigação.

2 DESENVOLVIMENTO DA PESQUISA

Nesse capitulo é apresentada uma visão geral da pesquisa desenvolvida considerando a classificação da pesquisa e o método de pesquisa empregado.

2.1 Considerações iniciais

Metodologia é o conjunto de métodos, de técnicas e de procedimentos com a finalidade de viabilizar a execução da pesquisa, obtendo como resultado um novo processo, produto ou conhecimento (JUNG, 2004; PRADANOV et al., 2013).

O restante deste capítulo está organizado da seguinte maneira. O tipo de pesquisa, considerando a natureza, os objetivos, a abordagem do problema e o enfoque dos procedimentos, é abordado na Seção 2.2. O método de pesquisa seguido para a realização deste trabalho é apresentado na Seção 2.3.

2.2 Tipo de pesquisa

As pesquisas podem ser classificadas quanto a (GERHARDT; SILVEIRA, 2009; JUNG, 2004):

a) **Natureza**. Uma pesquisa pode ser básica (gerar novos conhecimentos, úteis para o avanço da Ciência, sem aplicação prática prevista) ou aplicada (gerar conhecimentos para aplicação prática, dirigidos à solução de problemas específicos). Este trabalho pode ser caracterizado como **pesquisa aplicada**, pois uma abordagem de reestruturação de sistemas de software para ser aplicada na prática é proposta;
b) **Objetivos**. Uma pesquisa pode ser exploratória (proporcionar mais familiaridade com o tema), descritiva (descrever fatos e fenômenos de determinada realidade sem interferência do pesquisador no tratamento) ou explicativa (identificar fatores que contribuem ou determinam a ocorrência de um evento). Este trabalho pode ser classificado como **pesquisa descritiva**, pois, na sua condução, foram realizadas medições e avaliações de medidas de software;
c) **Abordagem**. Uma pesquisa pode ser quantitativa (resultados podem ser quantificados, sendo centrada na objetividade) ou qualitativa (não tem foco em avaliar o relacionamento numérico do estudo, mas a compreensão em profundidade de algum fato). Este trabalho pode ser classificado como **pesquisa quantitativa**, pois o resultado

do processo de reestruturação pode ser expresso de forma numérica. Assim, podem ser verificados ganhos ou perdas obtidas na condução desse processo de forma objetiva;

d) **Procedimentos**. Uma pesquisa pode ser caracterizada como:
   - **Pesquisa pré-experimental**. Pesquisa em que o experimento é executado sem o controle adequado das variáveis do ambiente que podem impactar no tratamento;
   - **Pesquisa experimental**. Pesquisa que segue um planejamento rigoroso controlando as variáveis e os fenômenos que possam atuar sobre o tratamento;
   - **Pesquisa bibliográfica**. Pesquisa feita a partir do levantamento de referências teóricas publicadas (*e.g.* livros e artigos);
   - **Pesquisa documental**. Pesquisa semelhante à pesquisa bibliográfica, mas possui fonte de pesquisa mais diversificada (*e.g.* jornais, cartas, pinturas, filmes e relatórios de empresas);
   - **Pesquisa de campo**. Pesquisa que reúne informações e/ou conhecimentos acerca de um problema, para o qual se procura uma resposta, ou de uma hipótese, que se queira comprovar ou descobrir novos fenômenos ou as relações entre eles;
   - **Pesquisa *ex-post-facto***. Pesquisa com o objetivo de investigar possíveis relações de causa e efeito entre um fato e um fenômeno que ocorre posteriormente. Os dados são coletados após o acontecimento do fato estudado;
   - **Pesquisa de levantamento**. Pesquisa exploratória e descritiva da população que pode considerar a análise de uma parcela ou de toda a população para determinar seu comportamento em relação a algum fenômeno;
   - **Pesquisa como *survey***. Pesquisa que busca informações diretamente com um grupo de interesse relacionado com o assunto estudado;
   - **Estudo de caso**. Pesquisa realizada com uma ou poucas entidades por vez (*e.g.* empresa e instituição pública) para conhecer e compreender detalhadamente o objeto de estudo;
   - **Pesquisa participante**. Pesquisa que responde às necessidades para população analisada e que se caracteriza pelo envolvimento do investigador com população investigada;
   - **Pesquisa ação**. Pesquisa que acontece quando os pesquisadores e os participantes representativos do problema estão envolvidos de modo cooperativo em uma ação para a resolução de um problema coletivo;
   - **Pesquisa etnográfica**. Pesquisa que estuda e busca entender a cultura de comunidades e grupos sociais;

- **Pesquisa etnometodológica**. Pesquisa que visa compreender como as pessoas constroem ou reconstroem a sua realidade social.

Este trabalho pode ser classificado como **pesquisa pré-experimental**, pois a arquitetura do sistema de software é avaliada em dois momentos distintos, antes de aplicar a abordagem de reestruturação (estrutura original) e após sua aplicação (estrutura sugerida) configurando um pré e pós-teste sobre um único grupo sem controlar as variáveis do ambiente (GIL, 2002).

2.3 Método de pesquisa

O método de pesquisa empregado para o desenvolvimento deste trabalho é composto por três fases e dez atividades (Figura 2.1):

a) **Embasamento da Reestruturação**. Nessa fase, foram criadas as bases para a construção da abordagem de reestruturação. O foco foi construir e solidificar conhecimento sobre medidas de software e técnicas/heurísticas de Inteligência Artificial. Nessa fase, como fruto do aprimoramento do conhecimento relacionado aos pontos estudados, foram definidos alguns aspectos para a abordagem desenvolvida. Essa fase é constituída por três atividades:
   - **Relacionar medidas internas e externas**. Nessa atividade, foi realizada uma Revisão Sistemática da Literatura (RSL), a qual reúne estudos sobre o mesmo tema, auxiliando no conhecimento do estado da arte e na condução de trabalhos futuros de um tema específico. A condução dessa RSL propiciou direcionamento ao desenvolvimento desta pesquisa, indicando as medidas externas de software existentes na literatura, os atributos internos e medidas internas que impactam as medidas externas e se elas refletem a qualidade de sistemas de software. Nessa RSL, foram identificadas medidas internas relacionadas com características de qualidade externa, inclusive manutenibilidade, característica que a abordagem elaborada para reestruturar sistemas de software visa aprimorar;
   - **Escolher medidas internas**. Nessa atividade, com base no conjunto de medidas obtido na atividade anterior, foram escolhidas medidas de acoplamento e de coesão utilizadas para elaborar uma abordagem para melhorar a estrutura interna de sistemas de software (organização de classes entre pacotes). Essa melhoria consiste na reestruturação desses sistemas para melhorar a sua qualidade interna (aumentar coesão e reduzir acoplamento entre os pacotes). Essas medidas atendem aos seguintes requisitos: i) ser amplamente difundida e aceita na academia; ii) possuir artigos

recomendando sua utilização para medir acoplamento ou coesão de sistemas de software; e iii) ser relacionada com a manutenibilidade de sistemas de software ou adequar-se às necessidades da realização deste trabalho. As medidas escolhidas foram organizadas em dois grupos, para evitar que as medidas responsáveis pela avaliação do resultado da reestruturação fossem influenciadas pelas medidas utilizadas durante a reestruturação. Um grupo de medidas foi utilizado como fonte de dados, para diagnosticar o estado da estrutura do sistema de software em relação aos atributos acoplamento e coesão. O valor dessas medidas é utilizado para verificar se as alterações realizadas durante a reestruturação melhoram a qualidade da estrutura interna do software a cada classe movimentada entre pacotes. Esse grupo de medidas, denominado **grupo de reestruturação**, é responsável por guiar a reestruturação. Outro grupo de medidas foi utilizado para aferir e ratificar se a reestruturação realizada proporcionou ganhos estruturais ao sistema de software. Esse grupo, denominado **grupo de avaliação**, é responsável por avaliar os ganhos obtidos pela reestruturação. As medidas presentes nesses dois grupos são distintas para garantir que a avaliação não seja influenciada, pois, ao utilizar as mesmas medidas para realizar a reestruturação e a avaliação, ganhos estruturais certamente seriam obtidos;

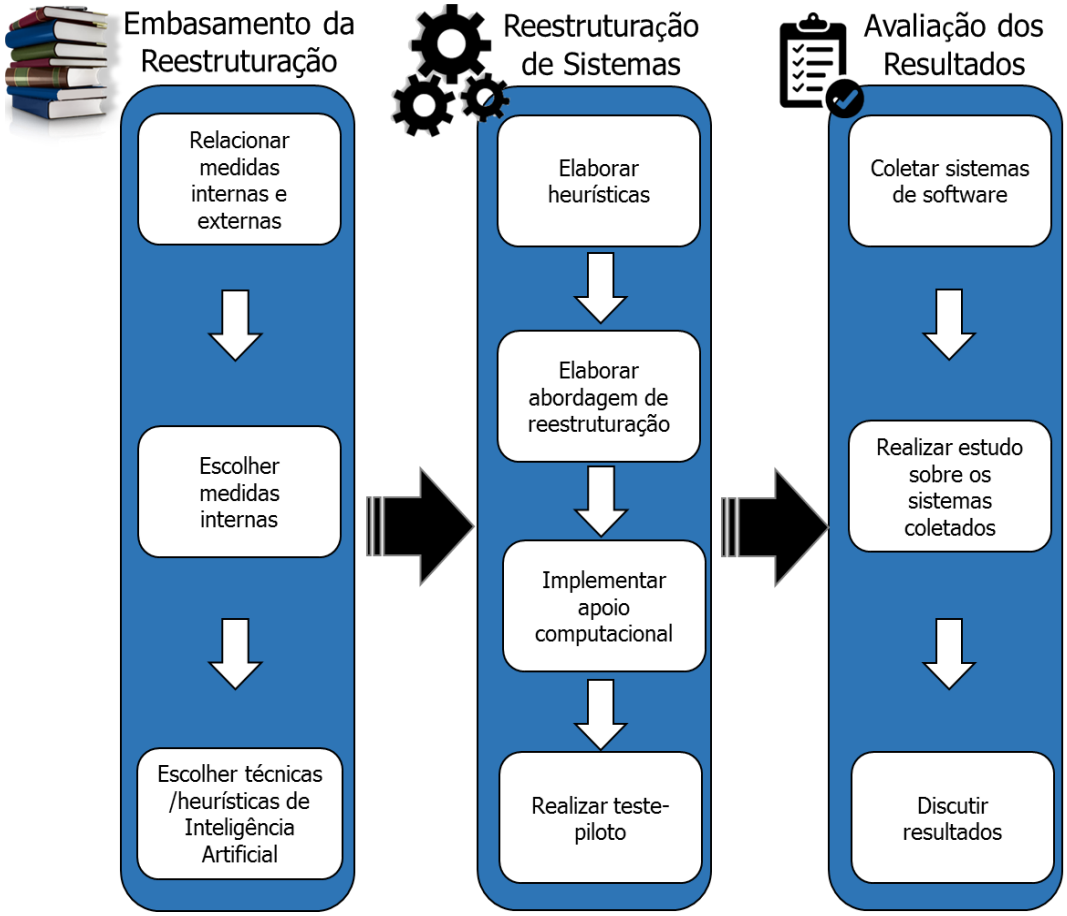
- **Escolher técnicas/heurísticas de Inteligência Artificial**. Técnicas/heurísticas de Inteligência Artificial (IA) são utilizadas para encontrar uma solução adequada para um problema em tempo viável. No contexto desta pesquisa, o objetivo da utilização de IA é maximizar as chances de sucesso e reduzir o tempo de execução e a subjetividade, que poderiam ser ocasionadas pela avaliação humana. Nessa atividade, foram estudadas técnicas e heurísticas de IA existentes na literatura. A heurística *Simulated Annealing* (SA) foi escolhida com base no conhecimento adquirido e nos trabalhos relacionados desta pesquisa para apoiar o processo de reestruturação;

b) **Reestruturação de Sistemas**. Nessa fase, foi elaborada e otimizada a maneira pela qual as alterações estruturais são conduzidas nos sistemas de software para aprimorar sua qualidade interna. Essa fase é constituída por quatro atividades:

- **Elaborar heurísticas**. Nessa atividade, foram elaborados três grupos de heurísticas para auxiliar na abordagem proposta para reestruturação. As **heurísticas de movimentação** constituem o cerne da abordagem de reestruturação, pois elas são responsáveis por determinar a probabilidade de uma classe ser movimentada entre os pacotes do sistema de software. As **heurísticas de decisão** definem a "melhor" classe a ser movimentada a cada iteração da abordagem. As **heurísticas de convergência**

definem se a estrutura da solução corrente deve ou não convergir para a estrutura de uma solução sugerida obtida ao longo da reestruturação;

Figura 2.1 - Método da Pesquisa

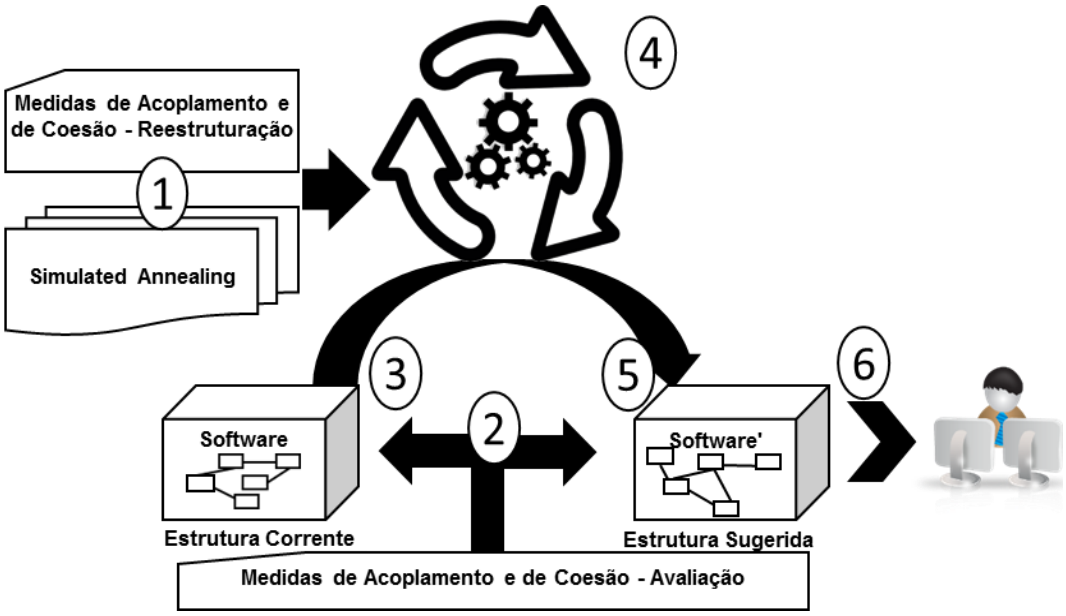


Fonte: Do autor (2017).

- **Elaborar abordagem de reestruturação**. Nessa atividade, há o desenvolvimento dos passos necessários para executar abordagem proposta para reestruturação. Inicialmente, essa abordagem utiliza a heurística SA e as medidas do grupo de reestruturação para realizar a reestruturação (**1** - Figura 2.2). As medidas do grupo de avaliação são utilizadas no início e no final da reestruturação para avaliar as estruturas internas corrente e sugerida, respectivamente (**2** - Figura 2.2). O sistema de software a ser reestruturado (**3** - Figura 2.2) é a entrada para a abordagem proposta para reestruturação (**4** - Figura 2.2). Nesse ponto, a abordagem de reestruturação é utilizada para movimentar classes entre pacotes iterativamente, utilizando as medidas do grupo de reestruturação, a SA e as heurísticas elaboradas. O resultado da reestruturação é a sugestão de uma distribuição (reestruturação) de classes entre pacotes do sistema de software (**5** - Figura 2.2). Essa sugestão é apresentada ao

engenheiro de software que toma a decisão de aceitar ou não a estrutura interna sugerida, escolhendo a estrutura mais apropriada (**6** - Figura 2.2);

Figura 2.2 - Mapa Conceitual da Abordagem Proposta para Reestruturação de Sistemas de Software



Fonte: Do autor (2017).

- **Implementar apoio computacional**. Nessa atividade, foi desenvolvido um *plug-in* para a plataforma Eclipse IDE, que automatiza a abordagem proposta para reestruturação (**4** - Figura 2.2). Nesse *plug-in*, os passos dessa abordagem são executados automaticamente para obter um sistema de software com melhor qualidade estrutural. Além de realizar a reestruturação, esse *plug-in* facilita a compreensão da reestruturação, pois apresenta a estrutura original e a estrutura sugerida do sistema de software, utilizando Diagrama de Classes e Diagramas de Pacotes da UML (*Unified Modeling Language*) com os valores das medidas do grupo de avaliação;
- **Realizar teste-piloto**. Nessa atividade, foram realizados testes para avaliar se a abordagem proposta para reestruturação está adequada e gera um sistema de software mais coeso e menos acoplado. Esses testes foram realizados por meio do *plug-in*, realizando a reestruturação de sistemas de software Java de código aberto. Com base nos resultados obtidos nesses testes, foi possível aperfeiçoar a abordagem proposta para atingir seu objetivo adequadamente e otimizá-la para torná-la mais eficiente;

c) **Avaliação dos resultados**. Nessa fase, há atividades para avaliar e quantificar se a abordagem proposta para reestruturação aprimora a qualidade interna de sistemas de software. Essa fase é constituída por três atividades:

- **Coletar sistemas de software**. Nessa atividade, foi realizado o *download* dos sistemas de software utilizados para conduzir as avaliações da abordagem proposta para reestruturação. Esses sistemas foram coletados dos repositórios de software de código aberto Git[3] e Sourceforge[4] e escolhidos de acordo com os seguintes critérios: i) ser utilizado na avaliação de, pelo menos, um trabalho relacionado a esta pesquisa; ii) ser desenvolvido na linguagem de programação Java; iii) ter seu código disponibilizado; iv) possuir última modificação (*update*) a, no máximo, 4 anos; e v) possuir projeto para a plataforma Eclipse IDE;
- **Realizar estudo sobre os sistemas coletados**. Nessa atividade, foram realizadas duas avaliações. Na primeira avaliação, o objetivo foi avaliar a eficiência da abordagem proposta de reestruturação. Para isso, os sistemas de software coletados foram submetidos à abordagem por meio do *plug-in* desenvolvido e as medidas utilizadas, a quantidade de dependências, o tempo de reestruturação, a quantidade de classes movimentadas e a quantidade de movimentos realizados foram coletados. Com base nesses dados, uma análise estatística, composta por análise descritiva, teste de normalidade multivariada e teste de hipóteses multivariado, foi conduzida para verificar se as melhorias proporcionadas às variáveis (medidas de software) de forma correlacionada foram estatisticamente significativas. Na segunda avaliação, o objetivo foi comparar a abordagem proposta para reestruturação com as abordagens apresentadas por trabalhos relacionados. Para isso, foram identificadas ferramentas existentes que executam reestruturação de sistemas de software. Utilizando as ferramentas funcionais encontradas, os sistemas de software coletados na atividade anterior foram reestruturados. Com base na estrutura sugerida pelas ferramentas, um conjunto de medidas de software foi coletado e uma análise estatística multivariada foi conduzida para identificar qual abordagem sugeriu a melhor qualidade de estrutura interna para os sistemas analisados, quanto aos atributos coesão e acoplamento;
- **Discutir resultados**. Nessa atividade, os resultados obtidos com a realização dos estudos sobre os sistemas de software coletados foram interpretados e discutidos de forma crítica, para identificar quais os fatores da abordagem proposta levaram a obtenção desses resultados e porque isso aconteceu.

[3] http://git-scm.com/
[4] http://sourceforge.net/

2.4 Considerações finais

Este trabalho pode ser classificado como: uma pesquisa aplicada, pois propõe uma abordagem para reestruturação de sistemas de software; uma pesquisa descritiva, pois medições e avaliações do valor de medidas de software são executadas; uma pesquisa quantitativa, pois o resultado do processo de reestruturação pode ser expresso numericamente; e uma pesquisa pré-experimental, pois realiza um pré/pós-teste com a qualidade da estrutura interna de sistemas de software sem controlar as variáveis do ambiente.

O trabalho seguiu o método de pesquisa proposto, composto por três fases: Embasamento da Reestruturação, Reestruturação de Sistemas e Avaliação dos Resultados. Cada uma dessas fases é constituída de atividades, totalizando dez atividades necessárias para alcançar o objetivo desta pesquisa.

3 TRABALHOS RELACIONADOS

Na literatura, grande esforço tem sido despendido por diversos autores para realizar trabalhos que contribuam para a melhoria da qualidade de software, por meio de alterações estruturais no projeto de sistemas de software.

Em um trabalho (ABDEEN et al., 2013), evolução de um trabalho prévio (ABDEEN, 2009), foi apresentada uma abordagem de otimização multiobjetiva da estrutura do software, cujos objetivos foram: i) minimizar as dependências entre pacotes, transformando dependências entre pacotes em intrapacotes; ii) minimizar o acoplamento entre pacotes, reduzindo as dependências entre pacotes; iii) reduzir as dependências cíclicas entre pacotes; e iv) minimizar a modificação na modularização original. Para atingir esses objetivos, foi utilizado o algoritmo *Non-Dominated Sorting Genetic Algorithm* (NSGA-II), que explora o espaço de busca criando e evoluindo populações de estruturas candidatas, por meio de operadores genéticos e seleção. Para determinar a função objetivo desse algoritmo e avaliar a qualidade das soluções geradas, foram utilizadas as medidas *Inter-Packages Dependencies* (IPD), *Inter-Packages Conections* (IPC) *e Inter-Packages Connections e Dependencies* (IPCC e IPCD). Para avaliar essa abordagem, nesse trabalho, seu autor realizou um estudo comparativo entre essa abordagem e a abordagem apresentada no trabalho prévio desse autor em que a heurística *Simulated Annealing* (SA) foi utilizada. Para isso, as duas abordagens foram aplicadas em quatro sistemas de software de código aberto. Os resultados obtidos indicam que a abordagem que empregou NSGA-II foi melhor que a abordagem que empregou SA, pois a primeira consegue aprimorar a qualidade de distribuição dos pacotes com menor prejuízo à modularização original do sistema de software. Isso se deve ao fato que a abordagem com NSGA-II evita a união de pacotes e movimenta menor quantidade de classes entre pacotes, quando comparado a abordagem com SA.

Em outro trabalho (BAVOTA; MARCUS; OLIVETO, 2013), uma abordagem para decompor pacotes em unidades menores e mais coesas utilizando medidas estruturais e semânticas foi proposta. Essa abordagem analisa separadamente pacotes sugeridos por especialista como candidatos a reestruturação e sugere sua divisão em dois ou mais pacotes. Para realizar essa divisão, o pacote sugerido pelo especialista para ser reestruturado é analisado e os resultados são armazenados em uma matriz n x n, sendo n a quantidade de classes do pacote. As células dessa matriz são preenchidas com a probabilidade que as classes localizadas na linha e na coluna que se interceptam estejam em um mesmo pacote. Essa probabilidade é determinada com base nas medidas ICP (*Information-Flow-Based Coupling*) (LEE, 1995) e

CCBC (*Conceptual Coupling Between Classes*) (POSHYVANYK et al., 2009). Posteriormente, com base nessa matriz, os relacionamentos entre as classes são analisados e os grupos de classes relacionadas são extraídos. Caso sejam obtidos mais de um grupo, é proposta a divisão desse pacote em dois ou mais pacotes, de acordo com a quantidade de grupos identificados. A avaliação desse trabalho consistiu em um estudo de caso com cinco sistemas de software. As estruturas desses sistemas sugeridas por essa abordagem foram analisadas por estudantes com experiência nesses sistemas e sem conhecimento do objetivo do trabalho, para evitar vieses na avaliação. Como resultado, foi obtida a aprovação dos estudantes sobre a remodularização executada, indicando que a técnica proposta é eficiente para obter pacotes mais coesos e sem deteriorar o acoplamento. A ferramenta R3 apresentada na Seção 8.4 é uma evolução dessa pesquisa.

Em outro trabalho (ZANETTI et al., 2014), foi apresentada uma estratégia para a remodularização de sistemas de software baseada na técnica de refatoração *move refactoring* e apoiada pela heurística SA. Essa estratégia consiste em movimentar classes entre pacotes para gerar um sistema de software mais modularizado (alta coesão e baixo acoplamento entre pacotes). Para isso, uma classe do sistema de software é escolhida randomicamente e a probabilidade de movimentá-la para os pacotes aos quais ela está conectada é calculada, utilizando as dependências apresentadas pela classe. Esse processo é empregado iterativamente até que a heurística utilizada encontre a "melhor" solução no espaço de busca de acordo com sua função objetivo, que se refere a quantidade de dependências entre os pacotes do sistema de software analisado. A avaliação desse trabalho foi realizada por meio de um estudo de caso com 39 sistemas de software, obtendo, em média, 77% de aperfeiçoamento na modularização dos sistemas de software em relação as suas estruturas originais. Além disso, foram comparadas as sugestões de movimentação de classes propostas por essa estratégia e as feitas por especialistas em conduzir remodularizações. O resultado obtido revelou baixo *Precision* e alto *Recall*, ou seja, divergência entre o apresentado pelos especialistas e o sugerido pela estratégia proposta.

Outro trabalho (PINTO; COSTA, 2014), apresenta uma abordagem para 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 CBO (*Coupling Between Objects*) 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 todas as classes do sistema de software. Para avaliar esse trabalho, foi utilizado um estudo de caso

com cinco sistemas de software. O resultado dessa avaliação indica que houve melhoria na modularização do software utilizando a abordagem proposta. Além disso, foi avaliada se a ordem escolhida para analisar as classes do sistema de software interfere no resultado da reestruturação, obtendo resultado positivo.

Em outro trabalho, uma abordagem para decompor o pacote em unidades menores e mais coesas foi proposta (PALOMBA et al., 2015), sendo a complementação de um trabalho prévio que propôs a divisão de classes em unidades mais responsáveis (BAVOTA et al., 2012). A abordagem apresentada analisa o sistema de software sob dois pontos de vista: i) estrutural, utilizando a medida ICP; e ii) conceitual, utilizando a medida CCBC. Com base na soma dos valores dessas medidas, é obtido o acoplamento de cada classe do sistema; com a média do acoplamento das classes de cada pacote, é determinada a coesão dos pacotes do sistema de software. Posteriormente, com base nos valores da coesão de cada pacote, é determinado o 1º quartil, limite que delimita os 25% menores valores da amostra; os pacotes com coesão abaixo desse limite são definidos como pacotes candidatos a serem reestruturados. Esses pacotes são analisados e, quando conveniente, é sugerida sua divisão em unidades mais coesas. Para realizar essa análise e a sugestão de divisão, esse trabalho utiliza a abordagem de reestruturação apresentada pelo trabalho de Bavota (BAVOTA; MARCUS; OLIVETO, 2013). O resultado da avaliação da abordagem revela sua capacidade de aprimorar a qualidade do software ao gerar pacotes mais coesos sem deteriorar seu acoplamento. A ferramenta ARIES, apresentada na Seção 8.4, é a ferramenta que automatiza essa abordagem.

Assim como esses trabalhos, esta investigação aprimora a qualidade de sistemas de software por meio da reestruturação. Contudo, nesta investigação, foi realizada a reestruturação de sistemas de software com base na geração de novas soluções para o problema de forma determinística, que movimenta classes entre pacotes com passos bem definidos para aperfeiçoar simultaneamente o acoplamento e a coesão, aprimorando a modularização e a manutenibilidade desses sistemas. Além disso, a abordagem proposta não afeta a estrutura original dos pacotes dos sistemas de software, pois não adiciona, divide ou remove pacotes. Dessa forma, menor impacto é gerado à estrutura sugerida desses sistemas, tornando-a mais próxima da estrutura original e mais compreensível à equipe responsável por esses sistemas. Outros diferenciais dessa abordagem são: i) não necessitar de auxílio ou de indicações do usuário durante a reestruturação, pois as classes movimentadas são escolhidas com base em seus relacionamentos; ii) utilizar medidas de software amplamente difundidas na academia, ratificando os resultados apresentados pela estrutura sugerida; iii) ser apoiada pela heurística de otimização combinatória *Simulated Annealing*, para identificar a “melhor” configuração

estrutural possível em tempo viável para o sistema de software; e iv) não alterar a estrutura de pacotes do sistema reestruturado, reduzindo os impactos gerados à estrutura do sistema. Isso facilita que o mantenedor compreenda a estrutura do sistema após a reestruturação, pois possui uma estrutura próxima da estrutura original, visto que somente as classes tiveram sua localização modificada.

Na Tabela 3.1, é apresentada uma sumarização dos trabalhos relacionados citados. Para cada trabalho é apresentado: i) se o trabalho utiliza uma técnica/heurística de Inteligência Artificial; ii) se o trabalho possui apoio computacional que automatize a abordagem proposta; iii) se o trabalho faz uso de medidas de software na condução da reestruturação; e iv) qual o mecanismo utilizado pela abordagem do trabalho para encontrar oportunidades de reestruturação, ou seja, determinar o ponto em que se deve agir para aprimorar o sistema de software (*e.g.* medidas de software e indicação do mantenedor).

Tabela 3.1 - Sumarização dos Trabalhos Relacionados

Trabalho	**Utiliza Inteligência Artificial**	**Possui Apoio Computacional**	**Utiliza Medidas de Software Difundidas**	**Modo de Encontrar Oportunidades de Reestruturação**
(ABDEEN et al., 2013)	Sim	Não	Não	Medidas de Software
(BAVOTA; MARCUS; OLIVETO, 2013)	Não	Sim	Não	Indicação humana
(ZANETTI et al., 2014)	Sim	Sim	Não	Escolha Randômica
(PINTO; COSTA, 2014)	Não	Não	Sim	Medidas de Software
(PALOMBA et al., 2015)	Não	Sim	Sim	1º quartil dos valores da coesão
Abordagem Proposta Neste Trabalho	Sim	Sim	Sim	Medidas de Software

Fonte: Do Autor (2017).

4 MEDIDAS DE SOFTWARE

Nesse capitulo é apresentada uma visão geral das medidas de software e um detalhamento das medidas empregadas nesta investigação.

4.1 Considerações iniciais

Medição é um elemento-chave em qualquer processo de engenharia (PRESSMAN; MAXIM, 2016), não sendo diferente na área de Engenharia de Software. Nessa área, o processo de medição provê meios para avaliar um conjunto de características/atributos do sistema de software (FENTON; NEIL, 2000, HONGLEI; WEI; YANAN, 2009), sendo impraticável realizar essa avaliação sem o auxílio de medidas (BRYTON; ABREU, 2008). Além disso, essas medidas podem ser utilizadas como fonte de dados para construção de uma base histórica de medição, sendo utilizada em projetos futuros da mesma organização como padrão para delinear desenvolvimento (SOMMERVILLE, 2015).

A medição fornece bases sólidas para a tomada de decisão pelos engenheiros de software (GYIMOTHY, 2009), aumentando seu entendimento sobre o processo, o sistema de software e o ambiente em que esse sistema está inserido. Portanto, medidas auxiliam no gerenciamento e no aprimoramento da qualidade do processo, do projeto e do produto (sistema de software) (FOCUS, 2015). Além disso, a utilização de medidas é motivada pelo fato dessas medidas propiciarem dados que possibilitam saber se sistemas de software desenvolvidos possuem, por exemplo, modularidade, reusabilidade e analisabilidade, prezando para que esses sistemas possuam “boa” estrutura e sejam fáceis de manter.

O restante deste capítulo está organizado da seguinte maneira. A distinção entre os termos métricas, medidas e medição é discutida na Seção 4.2. A classificação das medidas de software é apresentada na Seção 4.3. Definições “clássicas” das medidas escolhidas de acoplamento e de coesão para compor a abordagem proposta para reestruturação são abordadas nas Seções 4.4 e 4.5, respectivamente.

4.2 Métricas, medidas e medição

Na literatura, ainda há “confusão” no emprego dos termos **métricas**, **medidas** e **medição**. Entretanto, tais termos possuem diferenças sutis (PRESSMAN; MAXIM, 2016); para algumas pessoas (IEEE 610.12, 1990), **métrica** é definida como a escala e o método utilizados

no processo de medição, enquanto, para outras pessoas (SOMMERVILLE, 2015), **métrica** é uma característica do sistema de software, da sua documentação ou do seu processo de desenvolvimento que pode ser objetivamente mensurada. Essa nomenclatura foi suprimida na norma ISO/IEC 25000, que apresenta os termos (ISO/IEC 25000, 2014; IEEE 610.12, 1990):

a) **Medida (*measure* - substantivo)**. Variável para a qual é atribuído um valor como resultado do processo de medição;
b) **Medida (*measure* - verbo)**. Refere-se ao processo de realizar a medição, consistindo em um conjunto de operações necessárias para determinar o valor de uma medida;
c) **Medição (*measurement*)**. Processo de realizar um conjunto de operações para atribuir valor para uma medida.

Neste trabalho, foram utilizadas as definições estabelecidas pela norma ISO/IEC 25000, por ser a norma em vigor.

4.3 Classificação das medidas

As medidas de software podem ser organizadas em três grupos (FENTON; NEIL, 2000; KAN 2002; HONGLEI; WEI; YANAN, 2009; SOMMERVILLE, 2015; PRESSMAN; MAXIM, 2016):

a) **Medidas de produto** são utilizadas para medir atributos internos do sistema de software (*e.g.* coesão, tamanho e acoplamento). Essas medidas podem ser categorizadas em dois conjuntos (NICOLAESCU et al., 2015): i) medidas **estáticas**, coletadas sobre representações do sistema (*e.g.* projeto e documentação); e ii) medidas **dinâmicas**, coletadas com o sistema em execução, as quais são dependentes do ambiente no qual são utilizadas;
b) **Medidas de processo** são utilizadas por empresas para avaliar e aprimorar o processo de desenvolvimento. Por exemplo, tempo para consertar um erro e eficácia de remoção de defeitos ao longo do desenvolvimento;
c) **Medidas de projeto** descrevem características do projeto e de sua execução, por exemplo, quantidade de desenvolvedores, custo e produtividade (KAN, 2002).

Sob outro ponto de vista, para avaliar a qualidade do produto ao longo do seu ciclo de vida, as medidas podem ser classificadas em (ISO/IEC 25000, 2014):

a) **Medidas de qualidade interna**, referentes à medição de um atributo estático de sistemas de software relacionado à sua arquitetura (*e.g.* quantidade de linhas de código

e nível de acoplamento). As medidas internas possuem uma observação importante quanto à sua utilidade: caso elas não possuam evidência de relacionamento com algum atributo de qualidade externa de software, elas têm pouco valor (KAN, 2002; ISO/IEC 25000, 2014). Essas medidas podem ser utilizadas ao longo de todo o ciclo de vida, estando voltadas principalmente para avaliar estágios do desenvolvimento;

b) **Medidas de qualidade externa**, derivadas da avaliação do comportamento de sistemas de software, sendo utilizadas principalmente para avaliar esses sistemas em teste em um ambiente simulado (*e.g.* quantidade de defeitos encontrados em um teste e tempo de duração de uma tarefa de manutenção);

c) **Medidas de qualidade em uso**, referentes ao quanto os sistemas de software atendem as necessidades do usuário, em um contexto específico (*e.g.* eficiência e satisfação). Essas medidas avaliam o quanto os usuários podem atingir dos seus objetivos em determinados ambientes.

Apesar de ser um assunto não muito recente na área de Engenharia de Software, por ter seu início datado por volta dos anos 70 (HONGLEI; WEI; YANAN, 2009), as medidas de software têm ganhado relevância nos últimos anos, pois sua utilização tornou-se fator essencial para o sucesso no desenvolvimento de sistemas de software (HONGLEI; WEI; YANAN, 2009). Por isso, grande esforço tem sido despendido para evoluir o conhecimento nessa área, principalmente com base em suítes de medidas propostas na literatura amplamente utilizadas (*e.g.* CHIDAMBER e KEMERER (CK) (CHIDAMBER; KEMERER, 1994), *Metrics for Object Oriented Design* (MOOD) (ABREU; CARAPUÇA, 1994) e LORENZ e KIDD (LK) (LORENZ; KIDD, 1994)).

No escopo deste trabalho, algumas medidas de software foram escolhidas para serem utilizadas na abordagem proposta para reestruturação, avaliando os atributos acoplamento e coesão. A escolha foi baseada em medidas que (i) fossem amplamente difundidas na academia, (ii) possuíssem artigos recomendando-as para medir acoplamento ou coesão e (iii) estarem relacionadas à característica de qualidade manutenibilidade.

4.4 Medidas de acoplamento

Medidas de acoplamento expressam a interdependência entre módulos em um sistema de software (AL DALLAL, 2013), sendo consenso na literatura que esses sistemas com “boa” qualidade interna devem possuir baixo nível de acoplamento (CHIDAMBER; KEMERER, 1994; CHAE et al., 2000; CHEN et al., 2002; BAVOTA; MARCUS; OLIVETO, 2013; AL

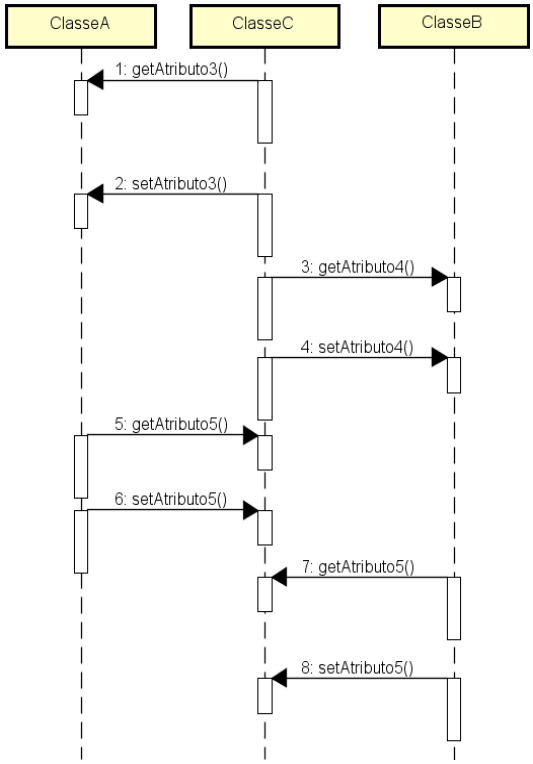
DALLAL, 2013). Nesta investigação, de acordo com os critérios de seleção especificados na Seção 2.3 e em uma Revisão Sistemática da Literatura (SANTOS et al., 2016a), cinco medidas de acoplamento foram selecionadas para compor a abordagem proposta para reestruturação:

a) **MPC (*Message Passing Coupling*)**. Analisa a complexidade da comunicação entre classes (LI; HENRY, 1993). Para determinar seu valor para a classe, é contabilizada a quantidade de métodos chamados definidos em outras classes (MC_{ca} - métodos chamados pela classe analisada), indicando o quanto os métodos locais são dependentes de métodos de outras classes. A medida MPC para o sistema pode ser calculada por:

$$MPC = \sum_{i=0}^{n} MC_{ca}$$

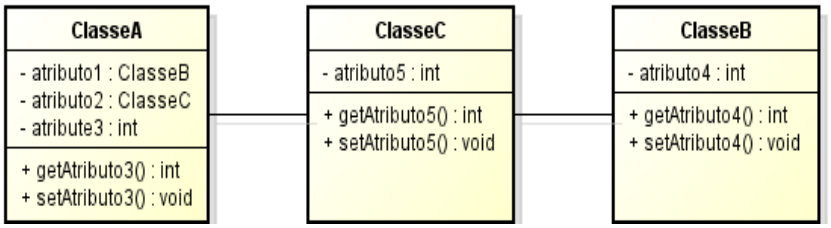
sendo `n` a quantidade de classes existentes no sistema. Por exemplo, baseado no diagrama de sequência apresentado na Figura 4.1 e no diagrama de classes apresentado na Figura 4.2 e, o valor da medida MPC para a classe `ClasseC` é igual a `4`, pois essa classe pode realizar chamadas a quatro métodos definidos em outras classes, os métodos `getAtributo3()` e `setAtributo3()` da classe `ClasseA` e os métodos `getAtributo4()` e `setAtributo4()` da classe `ClasseB`. A medida MPC foi escolhida para compor a abordagem proposta para reestruturação, pois provê informação útil para analisar sistemas de software (AGGARWAL et al., 2006), possuindo medição do acoplamento mais refinada que as medidas da suíte CK (SUBRAMANYAM; KRISHNAN, 2003). Além disso, é uma medida relacionada à característica de qualidade manutenibilidade (SANTOS et al., 2016a);

Figura 4.1 - Diagrama de Sequência do Exemplo para as Medidas MPC, CBO e RFC



Fonte: Do Autor (2017).

Figura 4.2 - Diagrama de Classes do Exemplo para as Medidas MPC, CBO e RFC



Fonte: Do Autor (2017).

b) **CBO (*Coupling Between Objects*)**. Contabiliza a quantidade de classes acopladas à classe analisada (CA - classe acoplada à classe analisada), considerando como acoplamento quando uma classe pode iniciar a execução de um método declarado na classe analisada ou acessar um atributo existente na classe analisada (CHIDAMBER; KEMERER, 1994). O valor da medida CBO para o sistema pode ser calculada por:

$$CBO = \sum_{i=0}^{n} CA_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. No exemplo da Figura 4.2, tem-se que o valor da medida CBO para as classes `ClasseA` e `ClasseB` é igual a `1`, pois essas classes relacionam-se somente com a classe `ClasseC`. O valor da medida CBO

para a classe `ClasseC` é igual a `2`, pois essa classe relaciona-se com as classes `ClasseA` e a `ClasseB`. Os autores dessa medida justificam sua existência pela necessidade de conhecer o valor do acoplamento para reduzi-lo, aumentando a modularidade e a reusabilidade. A medida CBO foi escolhida para compor a abordagem proposta para reestruturação, pois está relacionada à característica de qualidade manutenibilidade (SANTOS et al., 2016a), sendo uma medida amplamente difundida na comunidade científica por pertencer a suíte CK;

c) **RFC (*Response for Class*)**. Mensura a quantidade de métodos de uma classe que podem ter sua execução iniciada por meio de uma chamada recebida de outra classe do sistema de software (CHIDAMBER; KEMERER, 1994). Portanto, a medida RFC indica o potencial de comunicação da classe com o restante do sistema de software, sendo um "aprofundamento" da medida CBO, visto que, ao invés de contabilizar quantas são as classes conectadas à classe analisada, contabiliza a quantidade de caminhos conectados à classe analisada. Seu valor é determinado pela soma da quantidade de métodos existentes na classe analisada (M_{ca}) e a quantidade de métodos remotos (externos a classe analisada) que possam ser chamados diretamente por essa classe (MC_{ca}). O valor da medida RFC para o sistema pode ser calculada por:

$$RFC = \sum_{i=0}^{n} M_{ca} + MC_{ca}$$

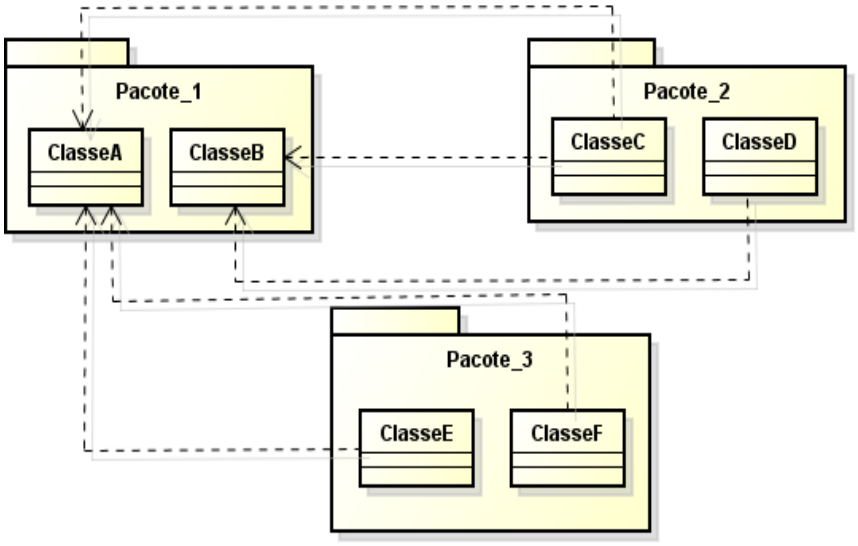
sendo `n` a quantidade de classe existente no sistema. No exemplo da Figura 4.2, tem-se que o valor da medida RFC para as classes `ClasseA` e `ClasseB` é igual a `4`, pois elas possuem dois métodos e podem chamar mais dois métodos da classe `ClasseC`. O valor de RFC para a classe `ClasseC` é igual a `6`, pois ela possui dois métodos e pode chamar mais quatro métodos das classes `ClasseA` (dois métodos) e `ClasseB` (dois métodos). Nesse exemplo, foi desconsiderado o conteúdo dos métodos (pois, em um cenário real, esses métodos poderiam chamar métodos de outras classes e aumentar o valor da medida RFC). A medida RFC foi escolhida para compor a abordagem proposta para reestruturação, pois está relacionada à característica de qualidade manutenibilidade (SANTOS et al., 2016a), justamente a característica que se deseja aprimorar com a reestruturação. Além disso, essa medida é amplamente difundida na comunidade cientifica por pertencer à suíte CK;

d) **Ca (*Afferent Coupling*)**. Analisa a responsabilidade de uma classe, revelando quantas classes externas ao pacote da classe analisada possuem dependências sobre ela (MARTIN, 1994). Assim, quanto maior a quantidade de dependências, maior o valor da medida Ca e maior a responsabilidade dessa classe, pois maior é a probabilidade de uma modificação na classe impactar em algum ponto do sistema de software. Para obter seu valor para a classe, é contabilizada a quantidade de classes externas ao pacote da classe analisada que dependem da classe analisada ($Cep_{ca}D_{ca}$). O valor da medida Ca para o sistema pode ser calculado por:

$$Ca = \sum_{i=0}^{n} Cep_{ca}D_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. Por exemplo, na Figura 4.3, o valor da medida Ca para a classe `ClasseA` é igual a `3`, pois as classes `ClasseC`, `ClasseE` e `ClasseF` dependem da classe `ClasseA`;

Figura 4.3 - Exemplo para as Medidas Ca e Ce



Fonte: Do Autor (2017).

e) **Ce (*Efferent Coupling*)**. Essa medida analisa o grau de dependência de uma classe em relação às demais partes do sistema de software (MARTIN, 1994). Seu valor para a classe é obtido pela contabilização da quantidade de classes que a classe analisada possui dependências incidentes fora do seu pacote ($Cep_{ca}DI_{ca}$). A medida Ce pode ser calculada por:

$$Ce = \sum_{i=0}^{n} Cep_{ca} DI_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. Analogamente à medida Ca, guardando as devidas diferenças, a medida Ce contabiliza a quantidade de classes de que se depende e não a quantidade de dependências. Por exemplo, na Figura 4.3, o valor da medida Ce para a classe `ClasseA` é igual a `0`, pois ela não depende de outras classes. As medidas Ca e Ce foram selecionadas para compor a abordagem proposta, pois são voltadas para a medição do acoplamento entre pacotes (DUCASSE et al., 2011), justamente a granularidade de medição trabalhada nesta investigação. Além disso, essas medidas são difundidas na literatura para medir esse tipo de acoplamento (ELISH, 2010).

4.5 Medidas de coesão

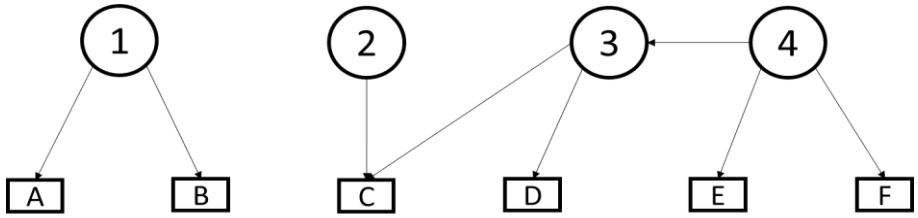
Medidas de coesão mensuram a força de associação entre elementos de um mesmo módulo/componente (STEVENS; MYERS; CONSTANTINE, 1974). Um sistema de software com "boa" qualidade interna deve ter alta coesão (CHIDAMBER; KEMERER, 1994; CHAE et al., 2000; CHEN et al., 2002; BAVOTA; MARCUS; OLIVETO, 2013; AL DALLAL, 2013), visto que um módulo/componente mais coeso é menos propenso a impactos por alterações em outras partes do sistema de software. Nesta investigação, de acordo com os critérios de seleção especificados na Seção 2.3 e em uma Revisão Sistemática da Literatura (SANTOS et al., 2016a), duas medidas de coesão foram selecionadas para compor a abordagem proposta para reestruturação:

a) **LCOM4 (*Lack of Cohesion in Methods*)**. Medida proposta como evolução às versões prévias de LCOM (1-3) (HITZ; MONTAZERI, 1995). Diferentemente das versões anteriores, essa medida baseia-se no princípio de grafos, em que os métodos são caracterizados como vértices e as dependências que um método possui são caracterizadas como arestas. Nesse contexto, uma dependência ocorre quando um método utiliza métodos ou acessa atributos (direta ou indiretamente) de outra classe do sistema de software. Finalizada a construção do grafo, o valor da medida LCOM4 para a classe é determinado pela quantidade de grafos desconexos (GD) existentes na classe, em que, quanto menor o valor obtido, melhor o estado do sistema. A medida LCOM4 para o sistema pode ser calculada pela fórmula:

$$\text{LCOM4} = \sum_{i=0}^{n} GD$$

sendo `n` a quantidade de classes do sistema. Por exemplo, no exemplo apresentado na Figura 4.4, os círculos de `1` a `4` representam métodos de uma classe, os retângulos de `A` a `F` representam atributos dessa classe e as setas entre métodos e atributos são as dependências existentes.

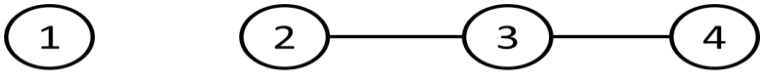
Figura 4.4 - Exemplo para a Medida LCOM4



Fonte: Do autor (2017).

Com base nesse exemplo (Figura 4.4), deve-se construir um grafo de dependências entre os métodos (Figura 4.5) para determinar o valor da medida LCOM4. Nesse grafo, há dois grafos desconexos, um formado pelo método `1` e o outro formado pelos métodos `2`, `3` e `4`, pois o método `1` não chama métodos ou acessa atributos em comum com os demais métodos. Portanto, o valor da medida LCOM4 é igual a `2`, pois existem dois grafos desconexos. Essa medida foi escolhida para compor o conjunto de medidas utilizadas na abordagem proposta, pois é uma medida recomendada para obter entidades mais modularizadas e não apresenta pontos negativos em relação ao seu resultado, ao contrário de outras versões dessa medida (DUCASSE et al., 2011);

Figura 4.5 - Grafo Gerado pelo Exemplo para a Medida LCOM4



Fonte: Do autor (2017).

b) **TCC (*Tight Class Cohesion*)**. Avalia a quantidade relativa de métodos diretamente conectados a classe (BIEMAN et al., 1995). Nesse contexto, pode-se compreender diretamente conectados como dois métodos utilizando atributos em comum. A medida TCC pode ser calculada por:

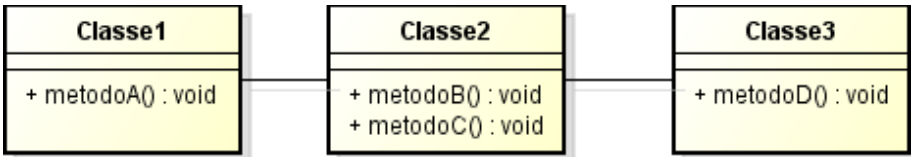
$$TCC\ (C) = \frac{NDC(C)}{NP(C)}$$

sendo C a classe analisada, NDC a quantidade de conexões diretas a classe e NP o total de conexões diretas e indiretas à classe. O valor de NP é determinado por:

$$NP(C) = \frac{N * (N - 1)}{2}$$

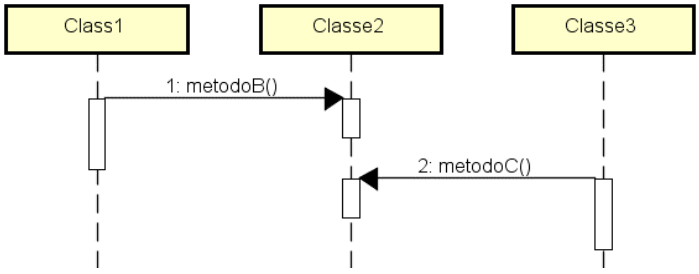
sendo N a quantidade de métodos existentes na classe analisada. Para exemplificar o cálculo do valor dessa medida, será utilizado como base o exemplo apresentado na Figura 4.6 e na Figura 4.7. Para obter o valor da medida TCC para a classe Classe2, inicialmente, calcula-se o valor de NDC. Visto que as classes Classe1 e Classe3 acessam diretamente a classe Classe2 (Figura 4.6), o valor de NDC para a Clsse2 é igual a 2. Além disso, calcula-se o valor de NP para a classe Classe2, utilizando (NP = 2 * (2 - 1) / 2), pois essa classe possui dois métodos; assim, o valor de NP é 1. Portanto, o valor da medida TCC para a classe Classe2 é igual a 2 (TCC = 2 / 1). A medida TCC foi escolhida para compor a abordagem proposta, pois é mais intuitiva que a medida LCC (*Loose Class Cohesion*) (JOSHI; JOSHI, 2010) e adequa-se as necessidades desta investigação.

Figura 4.6 - Diagrama de Classes para o Exemplo para a Medida TCC



Fonte: Do autor (2017).

Figura 4.7 - Diagrama de Sequência para o Exemplo para a Medida TCC



Fonte: Do autor (2017).

4.6 Considerações finais

Neste capítulo, foi apresentada a distinção entre os termos “métricas”, “medidas” e “medição”, visto que existe “confusão” na sua utilização. Com base nessas distinções, optou-

se por utilizar a nomenclatura sugerida na norma ISO/IEC 25000. Portanto, é empregado o termo “medida” para fazer referência ao valor atribuído a uma variável como resultado do processo de medição e não o termo “métrica”, como defendido nas demais definições.

Além disso, foram apresentadas formas de classificar as medidas de software em relação a qualidade interna e externa e/ou medidas de produto, de processo e/ou de projeto. Por fim, foi apresentado um conjunto de medidas de software relacionadas aos atributos acoplamento e coesão utilizados na abordagem proposta para reestruturação, apresentando suas definições, exemplos de cálculo e justificando o motivo da sua utilização.

5 *SIMULATED ANNEALING*

Nesse capitulo é apresentada uma visão geral da heurística *Simulated Annealing*, explicando seu funcionamento na teoria e também na prática por meio do seu pseudocódigo.

5.1 Considerações iniciais

Em Inteligência Artificial (IA), técnicas de programação são utilizadas por intermédio de máquinas para resolver problemas de modo semelhante ou mais eficiente que um especialista (RUSSELL, 1995; NIKOLOPOULOS, 1997). Com a aplicação de técnicas de IA neste trabalho, espera-se maximizar as chances de sucesso e reduzir o tempo de execução e a subjetividade, que poderia ser ocasionada pela avaliação humana. A utilização de técnicas/heurísticas de IA torna este trabalho interdisciplinar, pois une conceitos de duas áreas da Ciência da Computação (Inteligência Artificial + Engenharia de Software).

Com base nas técnicas/heurísticas de IA existentes na literatura e na leitura de trabalhos relacionados (BAVOTA; MARCUS; OLIVETO, 2013; ABDEEN et al., 2009; ZANETTI et al., 2014), a heurística de otimização combinatória *Simulated Annealing* (SA) foi selecionada para auxiliar na resolução do problema abordado nesta investigação. Outras técnicas de IA como Redes Neurais Artificiais (RNA) e Lógica *Fuzzy* foram analisadas. Contudo, essas técnicas não se adequaram as necessidades deste projeto, pois elas exigem informações para sua execução que não existe na academia ou no mercado, que se refere aos valores de referência para as medidas de software. Esses valores são essenciais para treinar a RNA e para definir intervalos de aceitação na Lógica *Fuzzy*.

A heurística SA foi proposta a partir da união dos conceitos de *annealing* (recozimento) e da resolução de problemas de otimização combinatória (VAN LAARHOVEN; AARTS, 1987; KIRKPATRICK; GELATT; VECCHI, 1983), tornando-a uma das heurísticas mais aplicáveis a problemas de otimização combinatória (AARTS; KORST; MICHIELS, 1997).

O conceito de *annealing* faz analogia ao processo utilizado na metalurgia para gerar metais com estrutura cristalina, em que se possui energia mínima, ou seja, são extremamente resistentes, pois seus átomos estão organizados de forma homogênea. Para atingir esse estado, o metal é aquecido a alta temperatura para seus átomos poderem se movimentar com liberdade. Posteriormente, esse metal é resfriado de forma lenta e gradual, até atingir o equilíbrio térmico e solidificar-se, para seus átomos organizarem-se em uma estrutura rígida e uniforme. A outra

face dessa heurística é a resolução de problemas de otimização combinatórios, cujo objetivo é maximizar/minimizar o custo de uma função dentro do espaço de busca.

O restante deste capítulo está organizado da seguinte maneira. Uma visão geral da heurística *Simulated Annealing* é apresentada na Seção 5.2. O pseudocódigo dessa heurística, para facilitar a compreensão do seu funcionamento e explicar alguns de seus pontos principais, é descrito na Seção 5.3.

5.2 A heurística

A heurística *Simulated Annealing* (SA) (KIRKPATRICK; GELATT; VECCHI, 1983) realiza seu processo de otimização em etapas, aprimorando gradativamente a função objetivo, assim como na metalurgia, em que se resfria o metal de forma gradual, reduzindo sua temperatura. Para realizar esse aprimoramento, SA faz uso da técnica de busca de vizinhos, gerando uma nova solução (solução vizinha) com base na solução atual (melhor solução encontrada até o momento). Posteriormente, essa heurística verifica se a solução vizinha é melhor que a atual, por meio do valor que cada uma apresenta em relação a função objetivo trabalhada (RUTENBAR, 1989), substituindo a solução atual pela solução vizinha, se a solução vizinha for melhor que a atual. Uma solução pior que a atual pode ser aceita para evitar mínimos locais no espaço de busca. Essa aceitação de soluções piores é maior no início do processamento da SA, fazendo com que o espaço de busca seja explorado. Entretanto, com o passar do tempo e com a obtenção de soluções que atendem melhor a função objetivo, essa heurística dificulta a aceitação de soluções piores que a atual, convergindo para um mínimo local, mas não necessariamente para um mínimo global.

A solução vizinha obtida em SA pode ser gerada aleatoriamente ou heuristicamente, recebendo apenas pequenas modificações em relação à solução atual. Portanto, a solução vizinha sempre está “perto” da solução atual, evitando mudanças bruscas de estado. Por isso, busca-se gerar apenas pequenas “perturbações” (alterações controladas ou aleatórias) na solução atual em busca de estados próximos que possuam uma estrutura para atender melhor a função objetivo.

5.3 O pseudocódigo

Para tornar a compreensão da heurística SA mais clara, seu pseudocódigo é apresentado na Figura 5.1. Nesse pseudocódigo, SA é executada até atingir uma solução com a

"temperatura" desejada ($T < T_{final}$) (linha 13), porém outras funções objetivo para analisar a solução e outros critérios de parada podem ser adotados de acordo com as especificidades do problema abordado. Outros pontos dessa heurística merecem destaques:

a) **Linha 4** - É construída aleatoriamente ou heuristicamente a solução (`proximo`) a partir da solução corrente (`candidato`). As estratégias empregadas para gerar essa solução podem ser as mais diversas, dependendo diretamente da natureza e das especificidades do problema abordado;
b) **Linha 5** - É calculada a variação da função objetivo (`energia(proximo)`) entre a solução (`próximo`) e a solução (`candidato`), quanto maior essa variação, menor a probabilidade da solução (`próximo`) ser aceita. Essa função objetivo determina o custo da solução analisada ("temperatura" na analogia à metalurgia) o qual revela se a solução (`próximo`) é "melhor" que a solução corrente (`candidato`);

Figura 5.1 - Pseudocódigo *Simulated Annealing*

```
1 candidato <- S0;    //candidato, "melhor" solução para o problema
2 T <- T0;            //Temperatura atual recebe temperatura inicial

3 repita
4   proximo <- vizinho(candidato); //"vizinho" é uma solução gerada
                                   //aleatoriamente ou heuristicamente
5   deltaE <- energia(proximo) - energia(candidato);

6   se deltaE <= 0 então
7     candidato <- proximo
8   senão
9     Numero_Randomico <- Rand [0..1];

10    se Numero_Randomico < exp(-deltaE/T) então
11      candidato <- proximo

12  T <- proximaTemperatura(T);
13até T < Tfinal

14retorna candidato;
```

Fonte: Do autor (2017).

c) **Linha 6** e **7** - Caso a solução (`proximo`) seja melhor (`deltaE <= 0`), ela torna-se a nova solução corrente (`candidato`);
d) **Linhas 8, 9, 10** e **11** - Em caso negativo, é calculada a probabilidade de aceitar uma solução (`proximo`) que possua o valor da função objetivo maior que a apresentada pela solução corrente (`candidato`), evitando mínimos locais no espaço de busca. Essa probabilidade é calculada com base na função de Boltzmann (`exp(-deltaE/T)`) (KIRKPATRICK; GELATT; VECCHI, 1983). Sendo assim, é aceita uma solução "pior" que a atual, caso o valor retornado por essa função seja maior que o valor obtido

randomicamente no intervalo `[0..1]`. Nota-se que a probabilidade de aceitar uma solução "pior" é diretamente influenciada pela variação entre as funções objetivo das soluções (`deltaE`). Por isso, a possibilidade de aceitar uma solução "pior" é maior no início da execução de SA (maior variação entre as soluções), reduzindo à medida que variações menores são alcançadas. Isso acontece, pois o resultado de `-deltaE/T` tende a diminuir, por causa da menor variação, reduzindo o expoente de $e^{\frac{-\text{deltaE}}{T}}$, e da probabilidade que esse número seja maior que o número aleatório obtido entre `0` e `1`, aceitando a solução (`proximo`). Por exemplo, a solução corrente (`candidato`) é igual a `100` e a solução (`próximo`) é igual a `50`, então o `deltaE` é igual `-50`. Logo, o $e^{\frac{-(-50)}{100}}$ é igual a `1,64` (para `e = 2,71`). Imaginando que essa solução (`proximo`) se tornou a solução corrente (`candidato`) e, na próxima iteração, obteve-se uma nova solução (`proximo`) com o valor da função objetivo igual a `40`, então o `delta` é igual a `-10`. Logo, tem-se $e^{\frac{-(-10)}{50}} = 1,22$ (para `e = 2,71`). Dessa maneira, percebe-se a redução da probabilidade de aceitação de soluções piores à medida que a variação entre as soluções reduz, convergindo para a solução do problema.

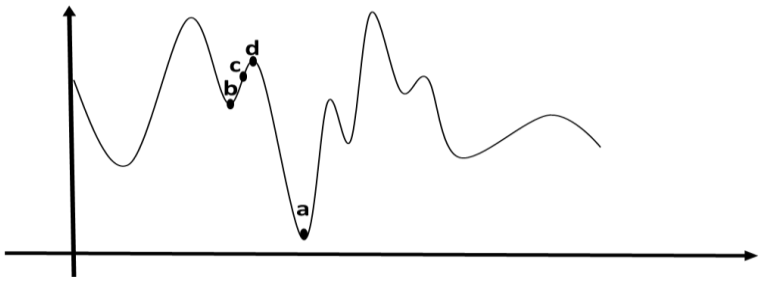
O principal diferencial proporcionado por SA, em relação as demais heurísticas de otimização combinatória, está relacionado entre as linhas 8 e 11, pois SA evita convergências a mínimos locais (ponto como menor valor para a função objetivo dentre os seus vizinhos mais próximos no espaço de busca) (RUTENBAR, 1989). Com isso, SA pode aceitar vizinhos com o valor da função objetivo pior que o do candidato atual na busca por encontrar o mínimo global (ponto com o menor valor dentre todos os pontos presentes no espaço de busca).

Dessa maneira, na Figura 5.2, ao chegar a um mínimo local (ponto **b**), SA pode aceitar soluções piores (pontos **c** e **d**) para alcançar um mínimo global (ponto **a**). Entretanto, essa heurística possui alguns pontos fracos, por exemplo, ser lenta (INGBER, 1993) e ter seus princípios fracamente embasados na matemática (CHARNES et al., 1989). A heurística SA possui diversos pontos que podem ser adaptados de acordo com o contexto do problema abordado, tais como, critérios de parada, geração das soluções vizinho e determinação da função objetivo para avaliar as soluções. Esse fato confere flexibilidade para SA e propicia que ela seja implementada de diversas maneiras para se adequar ao contexto do problema abordado.

Com base nessa visão geral da SA, nota-se que essa heurística pode ser adaptada para solucionar o problema de reestruturação abordado neste trabalho. Para isso, inicialmente, SA deve gerar uma solução vizinha da estrutura do sistema de software atual, a qual deve possuir

apenas algumas alterações estruturais em relação à estrutura atual desse sistema (solução atual). Com base na solução sugerida, SA verifica se essa solução atende de forma mais adequada a função objetivo do que a solução atual. Em caso positivo, a solução atual é descartada e a solução vizinha torna-se a atual; caso contrário, SA avalia a possibilidade de aceitar uma solução vizinha, mesmo sendo "pior" que a atual. Esse processamento se estende até não ser possível encontrar um vizinho melhor que o atual e o ponto que se acredita ser o mínimo global tenha sido atingido.

Figura 5.2 - Mínimos Locais e Mínimo Global no Espaço de Busca



Fonte: Do autor (2017).

5.4 Considerações finais

Neste capítulo, foi apresentada uma visão geral da heurística *Simulated Annealing*, abordando seu funcionamento, seu pseudocódigo e discutindo seus pontos principais. Essa heurística foi escolhida para compor o escopo dessa investigação com base no estudo das heurísticas e técnicas de IA existentes na literatura e na leitura dos trabalhos relacionados. O objetivo de utilizar SA no contexto desta pesquisa é tornar sua execução mais eficiente e menos subjetiva que a avaliação humana.

Os conceitos da SA estão embasados na analogia ao processo de recozimento do metal na metalurgia e na otimização de problemas combinatórios. Para realizar a otimização, essa heurística faz uso da técnica da busca de vizinhos. Assim, dada a situação inicial do problema (solução original), é gerado um vizinho, solução com algumas modificações em relação a melhor solução encontrada até o momento. Se o vizinho gerado atender a função objetivo de forma melhor que solução atual, ele se torna a solução atual na busca pela melhor solução. Em alguns casos, SA pode aceitar soluções "piores" para evitar mínimos locais e tentar alcançar a melhor solução global para o problema trabalhado. A geração de um vizinho e a avaliação de sua aceitação são executadas de forma cíclica até não encontrar novos vizinhos que melhor satisfaçam a função objetivo ou não atingir outro critério de parada (*e.g.* quantidade de iterações).

6 ARS - ABORDAGEM PARA REESTRUTURAÇÃO DE SOFTWARE

Nesse capitulo é apresentada uma visão geral abordagem desenvolvida para reestruturar sistemas de software.

6.1 Considerações iniciais

Neste capítulo, é apresentada uma abordagem para reestruturação de sistemas de software (ARS) para aprimorar a qualidade interna desses sistemas por meio da movimentação de classes entre pacotes (SANTOS; JUNIOR; COSTA, 2016b). Na literatura, existem trabalhos relacionados que apresentam abordagens semelhantes para melhorar a qualidade estrutural de sistemas de software, por exemplo, decompor pacotes em entidades menores, unir ou dividir pacotes ou movimentar classes entre pacotes (ABDEEN et al., 2013; PALOMBA et al., 2015). Entretanto, nesta investigação, a abordagem proposta realiza a reestruturação de sistemas de software de forma determinística por meio da movimentação de classes entre pacotes com passos bem definidos, utilizando a heurística *Simulated Annealing* (SA) e medidas de software difundidas na academia. Além disso, essa abordagem não alterar a estrutura de pacotes do sistema de software reestruturado e não sofre interferência de fatores humanos, excetuando os casos em que o mantenedor deseje definir restrições de movimentações de classes entre pacotes.

O restante deste capítulo está organizado da seguinte maneira. Adequação das medidas utilizadas neste trabalho é descrita na Seção 6.2. Adaptações feitas na heurística *Simulated Annealing* para adequá-la ao contexto dessa abordagem são expostas na Seção 6.3. Definições das heurísticas criadas para guiar as decisões durante a reestruturação são exibidas na Seção 6.4. Passos necessários para reestruturar sistemas de software de acordo com ARS são descritos na Seção 6.5. Exemplificação da utilização da abordagem é apresentada na Seção 6.6.

6.2 Adequação das medidas de software

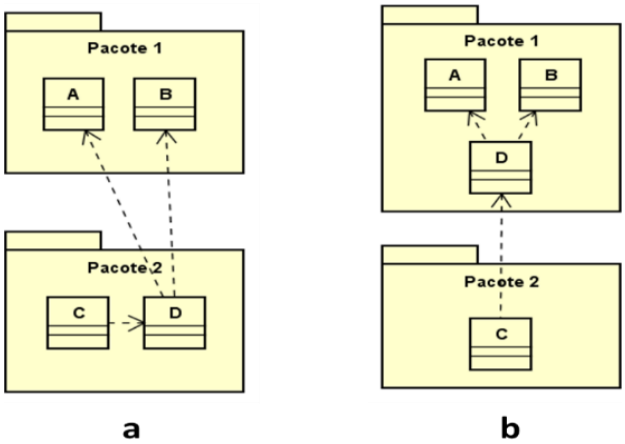
Para realizar a reestruturação utilizando a abordagem proposta, são utilizadas as medidas de software apresentadas no Capítulo 4. Essas medidas são utilizadas para medir o sistema de software durante a reestruturação e servir de base para determinar se a arquitetura do sistema está melhorando, por meio de seus valores. Entretanto, para serem utilizadas nessa abordagem, a granularidade de medição dessas medidas foi alterada da granularidade de classe para a granularidade de pacotes, exceto as medidas Ca e Ce (originalmente aferidas nessa

granularidade). Desse modo, considera-se como granularidade o nível de abstração em que as medidas utilizadas são medidas sobre o código, ou seja, quais tipos de relacionamentos são considerados (*e.g.* entre métodos, entre classes e entre pacotes). No contexto deste trabalho, as medidas estão na granularidade de pacotes, portanto são contabilizados somente os relacionamentos entre pacotes para determinar os valores das medidas para cada classe.

Essa alteração é motivada pelo fato que, ao movimentar classes entre pacotes, se a medição for realizada na granularidade de classes, os valores das medidas permanecem constantes, pois as dependências[5] nessa granularidade mantêm-se inalteradas. Dessa maneira, para detectar se um movimento realizado gerou impactos positivos ou negativos à estrutura do sistema, a medição deve ser realizada na granularidade de pacotes, pois é nessa granularidade que as dependências se alteram com o movimento de classes entre pacotes na ARS.

Para justificar e exemplificar essa alteração na granularidade, considera-se o exemplo apresentado na Figura 6.1. Nesse exemplo, a classe `D` possui mais dependências com o pacote `Pacote1` do que com seu pacote atual, o pacote `Pacote2` (Figura 6.1a). Portanto, pode ser "interessante" movimentar a classe `D` para o pacote `Pacote1` (Figura 6.1b).

Figura 6.1- Exemplo da Justificativa de Alteração na Granularidade das Medidas: (a) Antes do Movimento e (b) Depois do Movimento



Fonte: Do autor (2017).

Avaliando a quantidade de dependências da classe movimentada (classe `D`), pode ser avaliado se o movimento realizado gerou consequências positivas. Analisando essas dependências na granularidade de classes, observa-se que a classe `D` possui `3` dependências antes de efetuar o movimento (classes `A`, `B` e `C`), e após a efetivação do movimento, a quantidade de dependências mantém-se inalterada. Com a realização desse movimento, a modularização

[5] Indica a ocorrência de um relacionamento semântico entre dois ou mais elementos, em que um elemento é dependente de alguns serviços do elemento fornecedor.

visualmente foi aprimorada (Figura 6.1b), pois a classe `D` está localizada no pacote que possui mais dependências. Entretanto, essa melhoria na modularidade não foi detectada quantitativamente, pois, após o movimento a classe `D` possui as mesmas `3` dependências que possuía antes do movimento.

Por outro lado, avaliando essas dependências na granularidade de pacotes, em que somente são contabilizadas as dependências interpacotes, percebe-se que a quantidade de dependências da classe `D` foi reduzida de `2` (classes `A` e `B`) para `1` (classe `C`) com o movimento realizado. Com essa alteração na granularidade de medição, o aprimoramento proporcionado à modularização é perceptível visualmente e quantitativamente.

Desse modo, existe a necessidade de alterar a granularidade das medidas utilizadas para se detectar quantitativamente os ganhos proporcionados por ARS. Ressalta-se que as medidas Ca e Ce não foram alteradas em relação as suas definições originais descritas no Capítulo 4, pois originalmente são definidas nessa granularidade. A seguir, é apresentada a forma de calcular as demais medidas utilizadas com a granularidade de medição alterada. Essas medidas receberam o índice `P` (Pacote) para ressaltar que sua granularidade de medição foi alterada em relação a definição original:

a) **MPC_p** - Seu valor para a classe é obtido pela quantidade de métodos que a classe analisada chama de classes externas ao seu pacote ($MCep_{ca}$ - métodos chamados externos ao pacote da classe analisada). O valor da medida MPC_p para o sistema pode ser calculada por:

$$MPC_p = \sum_{i=0}^{n} MCep_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. Como base nessa medida, pode-se saber o quanto a classe analisada é dependente de outros pacotes e um alto valor da medida MPC_p indica que a classe possui muitas chamadas fora do seu pacote e pode estar "mal posicionada" na estrutura;

b) **CBO_p** - Seu valor é obtido pela quantidade de classes externas ao pacote da classe analisada acopladas à classe analisada ($CAep_{ca}$ - classe acoplada externa ao pacote da classe analisada), considerando como acoplamento quando uma classe possui dependências, chama métodos ou acessa atributos de outra classe. O valor da medida CBO_p para o sistema pode ser calculada por:

$$CBO_p = \sum_{i=0}^{n} CAep_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. A medida CBO_p indica o quanto uma classe está acoplada aos demais pacotes, em que quanto maior seu valor, pior, pois, se uma classe possui muitas dependências sobre classes fora do seu pacote, pode ser um indicativo que a classe está "mal posicionada" na estrutura;

c) **RFC_p** - Seu valor para a classe é obtido pela soma da quantidade de métodos existentes na classe analisada (M_{ca} - métodos da classe analisada) e da quantidade de métodos chamados por essa classe fora do seu pacote ($MCep_{ca}$ – métodos chamados externos ao pacote da classe analisada). A medida RFC_p pode ser calculada por:

$$RFC_p = \sum_{i=0}^{n} M_{ca} + MCep_{ca}$$

sendo `n` a quantidade de classes existentes no sistema. A medida RFC_p informa o nível de comunicação que uma classe possui 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;

d) **$LCOM4_p$** - Ao contrário das demais medidas, seu valor é contabilizado para cada pacote do sistema de software e não para cada classe, visto que se deseja saber a coesão do pacote e não da classe devido a alteração na granularidade de medição. Para obter o valor dessa medida, para cada pacote, é construído um grafo com base nas classes que o compõem e nas dependências entre elas, considerando como dependências métodos chamados e atributos acessados. Para construir esse grafo, as classes são consideradas como os vértices do grafo e as chamadas a métodos e acessos a atributos são como as arestas entre os vértices, levando em consideração somente as classes que compõe o pacote analisado. Construído o grafo, o valor da medida $LCOM4_p$ é determinado para o pacote analisado com base na quantidade de grafos desconexos (`GD`) obtidos. O valor da medida $LCOM4_p$ para o sistema de software pode ser calculada por:

$$LCOM4_p = \sum_{i=0}^{n} GD$$

sendo `n` a quantidade de pacotes do sistema. Se for obtido um grafo conexo, o valor da medida $LCOM4_p$ é igual a `1`; caso contrário, seu valor é determinado pela quantidade de grafos desconexos obtidos. Na situação em que o pacote é composto por somente uma classe, o valor da medida $LCOM4_p$ é igual a `1`. Dessa maneira, é determinado o quão coeso é cada pacote e a redução do valor da medida $LCOM4_p$ indica que os pacotes do sistema de software estão mais coesos, pois possuem menos conjuntos de classes que não se relacionam com as demais classes do pacote, formando grafos desconexos;

e) **TCC_p** - Seu valor é obtido com base na sua fórmula original $\left(TCC(C) = \frac{NDC(C)}{NP(C)}\right)$. Nessa fórmula, `NDC(C)` refere-se à quantidade de métodos que chamam métodos ou acessam atributos da classe analisada e, por causa da modificação na granularidade da medida TCC, `NDC` somente considera as chamadas internas ao pacote da classe analisada. `NP` é determinado pela sua fórmula padrão $NP(C) = \frac{N*(N-1)}{2}$, sendo `N` a quantidade de métodos existentes na classe analisada. A medida TCC_p pode ser calculada por:

$$TCC_p = \frac{NDC(C)}{NP(C)}$$

Dessa maneira, com base no valor da medida TCC_p, sabe-se o quão forte é a dependência da classe analisada com relação às demais classes do seu pacote. O aumento do valor da medida TCC_p indica que essa dependência foi fortalecida, pois a classe aumentou suas dependências com classes do seu pacote.

Essas medidas foram organizadas em dois grupos: i) **Grupo de Reestruturação**, composto pelas medidas MPC_p, CBO_p, RFC_p e TCC_p; e ii) **Grupo de Avaliação**, composto pelas medidas Ca, Ce e $LCOM4_p$. Foram adotados dois grupos distintos de medidas para evitar que as medidas responsáveis pela avaliação do resultado da reestruturação fossem influenciadas pelas medidas utilizadas durante a reestruturação. Outro ponto importante relacionado às medidas utilizadas foi a não adoção de pesos sobre elas em nenhum ponto de ARS, para não priorizar uma medida em relação as demais.

O grupo de reestruturação é utilizado para conduzir a reestruturação; por isso, esse grupo de medidas é utilizado a cada iteração de ARS para coletar dados e diagnosticar o estado atual da estrutura do sistema de software. O grupo de avaliação é utilizado para avaliar ganhos obtidos com a reestruturação; por isso, seus valores são apresentados ao usuário em dois momentos distintos, antes e após a reestruturação, para avaliar se as melhorias obtidas foram satisfatórias.

Por esse fato, as medidas do grupo de avaliação somente são aplicadas sobre a estrutura do sistema de software em dois momentos: i) ao iniciar a reestruturação, para coletar dados da estrutura original do sistema de software; e ii) ao finalizar a reestruturação, para coletar dados e avaliar os resultados obtidos na estrutura sugerida. Dessa forma, o processo de reestruturação é otimizado, pois evita-se que as medidas do grupo de avaliação sejam mensuradas desnecessariamente durante a reestruturação, visto que seus valores somente são necessários no início e no final da reestruturação para serem exibidos para o usuário. Portanto, mensurar essas medidas durante a reestruturação seria um desperdício de tempo e de recursos computacionais.

6.3 Adaptação da s*imulated annealing*

No contexto desta pesquisa, foram feitas algumas adaptações na heurística SA apresentada no Capítulo 5. Originalmente, SA verifica o valor do delta (diferença entre os valores da função objetivo da solução sugerida e da solução atual) para determinar a aceitação da solução sugerida. Nessa situação, normalmente, as variáveis utilizadas no problema são unidas por meio da função objetivo, determinando um valor único que indica o quanto a solução encontrada é "boa" em relação aos objetivos pretendidos. Entretanto, no contexto em que SA foi aplicada nesta pesquisa, somar, multiplicar, ponderar ou aplicar qualquer manipulação algébrica nos atributos utilizados para obter um valor único para a solução poderia levar a erros, pois o acoplamento e a coesão possuem sentido de crescimento contrários (o valor do acoplamento reduz e o valor da coesão aumenta ao serem aprimorados). Por isso, foram criados dois deltas (`deltaAcoplamento` e `deltaCoesão`) em analogia ao delta único existente na definição original dessa heurística.

O `deltaAcoplamento` indica a diferença entre os valores da função objetivo do acoplamento da solução sugerida (`FuncObjAcoplamentoSS`) e da solução atual (`FuncObjAcoplamentoSA`), determinado por:

```
deltaAcoplamento = FuncObjAcoplamentoSS - FuncObjAcoplamentoSA
```

A função objetivo do acoplamento indica a porcentagem de diferença que a solução analisada apresenta em relação a solução original, ou seja, o percentual que a solução analisada é melhor ou pior que a solução original. Nessa análise, valor negativo de `deltaAcoplamento` (valor da função objetivo da solução sugerida menor que o da solução

atual) representa aprimoramento no acoplamento do sistema de software, pois indica que o acoplamento da solução sugerida é menor do que o da solução atual, determinada por:

$$FuncObjAcoplamento = \frac{acoplamentoSA}{acoplamentoSO} * 100$$

sendo `acoplamentoSA` o valor do acoplamento da solução analisada (solução atual ou sugerida) e `acoplamendoSO` o valor do acoplamento do sistema de software original. Portanto, uma solução com valor de acoplamento menor apresenta percentual menor de acoplamento em relação a solução original. Assim, a diferença entre o valor do acoplamento da solução sugerida e o da solução atual (`deltaAcoplamento`) resulta em valor menor do que `0`. Isso é um indicativo que a solução sugerida é melhor do que a atual. Por exemplo, durante a primeira iteração da reestruturação, a solução original também é a solução atual. Portanto, dado valor do acoplamento original igual a `100`, o valor do acoplamento da solução atual é igual a `100` (`acoplamentoSA`), representando `100%` em relação à solução original. Se, na primeira iteração da reestruturação, a solução sugerida reduzir o valor de acoplamento para `80` (`acoplamentoSS`), representando `80%` em relação ao acoplamento original, então o valor de `deltaAcoplamento` obtido será igual a `-20` (`80 - 100`), ou seja, a solução sugerida aprimorou o acoplamento em relação à solução atual.

De maneira análoga, `deltaCoesão` indica a diferença entre os valores da função objetivo da coesão da solução sugerida (`FuncObjCoesãoSS`) e da solução atual (`FuncObjCoesãoSA`), determinado por:

$$deltaCoesão = FuncObjCoesãoSS - FuncObjCoesãoSA$$

A função objetivo da coesão indica a porcentagem de diferença que a solução analisada apresenta em relação a solução original, ou seja, o percentual que a solução analisada é melhor ou pior que a solução original. Nessa análise, valor positivo de `deltaCoesão` (valor da função objetivo da solução sugerida maior que o da solução atual) representa aprimoramento na coesão do sistema de software, pois indica que a coesão da solução sugerida é maior do que o da solução atual, determinada por:

$$FuncObjCoesão = \frac{coesãoSA}{coesãoSO} * 100$$

sendo `coesãoSA` o valor da coesão medida na solução analisada (solução atual ou sugerida) e `coesãoSO` o valor da coesão do sistema de software original. Portanto, uma solução com valor

de coesão maior apresenta percentual maior de coesão em relação a solução original. Assim, a diferença entre o valor da coesão da solução sugerida e o da solução atual (`deltaCoesão`) resulta em valor maior do que `0`. Isso é um indicativo de que a solução sugerida é melhor do que a atual. Por exemplo, durante a primeira iteração da reestruturação, a solução original também é a solução atual. Portanto, dado o valor da coesão da solução original igual a `100`, o valor da coesão da solução atual é igual a `100` (`coesãoSA`), representando `100%` em relação à solução original. Se, na primeira iteração da reestruturação, a solução sugerida aumentar o valor da coesão para `150` (`coesãoSS`), representando `150%` em relação à coesão original, então o valor de `deltaCoesão` obtido será igual a `50`, (`150 - 100`), ou seja, a solução sugerida aprimorou a coesão em relação à solução atual.

Por causa dessa adaptação no delta, a heurística SA necessitou ser modificada para verificar os ganhos proporcionados aos deltas definidos e determinar se a estrutura sugerida (vizinha) é melhor que a solução atual. Nesse ponto, para avaliar esses dois atributos (coesão e acoplamento) de forma independente, surgem quatro possibilidades:

a) Coesão e acoplamento foram aprimorados;
b) Somente a coesão foi aprimorada;
c) Somente o acoplamento foi aprimorado;
d) Coesão e acoplamento foram deteriorados.

Existem quatro possibilidades, pois, esses dois atributos não devem ser avaliados conjuntamente, visto que possuem sentido de crescimento opostos. Desse modo, é possível que em dado momento da reestruturação o ganho proporcionado a um atributo (coesão ou acoplamento) seja maior que a deterioração ocasionada ao outro atributo, logo a solução sugerida deve ser aceita, pois, no geral, o sistema melhorou. Para essa aceitação, com base nos deltas definidos, SA utiliza o **grupo de heurísticas de convergência** (Seção 6.4.3), cujo objetivo é determinar se a solução atual deve convergir para a solução sugerida. Utilizadas essas heurísticas, se a solução sugerida for considerada pior que a atual, SA a descarta; caso contrário, a solução sugerida tornar-se a solução atual. Desse modo, nas duas situações, a solução atual é retornada e utilizada como base para a geração de uma nova solução sugerida para tentar aprimorar a solução atual.

6.4 Definição das heurísticas

Nesta subseção, são apresentadas as heurísticas para servirem de regra e darem suporte à tomada de decisão ao longo da utilização da abordagem proposta de reestruturação.

6.4.1 Heurísticas de movimentação

As heurísticas do grupo de movimentação têm como objetivo determinar quais classes da estrutura atual de um sistema de software podem ser movimentadas entre os pacotes existentes. Essas heurísticas são um ponto crucial de ARS por encontrarem oportunidades de reestruturação, fato que não é uma tarefa trivial (DU BOIS; DEMEYER; VERELST, 2004). Esse grupo é composto por duas heurísticas:

a) `Heurística de Movimentação I (HMI) - Não mover classes de pacotes que contenham somente uma classe`.
   O objetivo é evitar o movimento da única classe de um pacote para outro pacote, ocasionando a existência de um pacote vazio e a consequente necessidade de excluí-lo. Isso pode gerar, ao final da reestruturação, a existência de um único pacote no sistema contendo todas as classes. Portanto, a existência dessa heurística justifica-se pela necessidade de evitar impactos à estrutura existente no sistema de software por causa da exclusão de pacotes, o que geraria mais modificações à estrutura do sistema, visto que são movimentadas classes entre os pacotes. Assim, excluir pacotes dificultaria a posterior compreensão da estrutura sugerida pelos mantenedores. Ao utilizar essa heurística, as classes "sozinhas" em um pacote recebem o valor `0` (zero) na probabilidade de movimentação (`PMov`). Dessa forma, é evitada a exclusão de pacotes não afetando a organização conferida aos pacotes do projeto. A `HMI` pode ser expressa por:

$$\text{HMI: } \forall\ P_i\ /\ |P_i| = 1, \text{então } \text{PMov}(C_{P_i}) = 0$$

   sendo P_i o enésimo pacote, $|P_i|$ a quantidade de classes do pacote `i` e C_{P_i} a classe do pacote `i`;

b) `Heurística de Movimentação (HMII) - Obter probabilidade de a classe ser movimentada para um pacote vizinho`.

Nesse contexto, pacote vizinho é o conjunto de pacotes que possuem classes que dependem ou são dependentes da classe analisada. Essa heurística é utilizada para obter o valor de PMov das classes não abrangidas pela heurística HMI. A base para o cálculo da probabilidade de movimentação é a quantidade de dependências que a classe apresenta com cada pacote vizinho, considerado como dependências chamadas a métodos e instanciação de atributos. Dessa maneira, PMov pode ser definida por:

$$PMov(C) = DepExterno(C) - DepInterno(C)$$

sendo DepExterno(C) a quantidade de dependências que a classe apresenta fora do seu pacote e DepInterno(C) a quantidade de dependências que a classe apresenta dentro do seu pacote. A HMII pode ser expressa por:

$$\text{HMII: } \forall P_i, \text{se } |P_i| > 1, \text{então } \forall C \in P_i \rightarrow \text{PMov}(C) = \text{DepExterno}(C) - \text{DepInterno}(C)$$

6.4.2 Heurísticas de decisão

As heurísticas do grupo de decisão têm por finalidade escolher a "melhor" classe a ser movimentada de um conjunto de classes com a mesma probabilidade de movimentação (mesmo valor para PMov). Para essa escolha, as heurísticas desse grupo são utilizadas sequencialmente; se o resultado da utilização de uma delas retornar somente uma classe, então as demais heurísticas são desconsideradas. Esse grupo é composto por cinco heurísticas:

a) Heurística de Decisão I (HDI) - Escolher classe com menor coesão.

Com essa heurística, é selecionada a classe com menor coesão em seu pacote atual para ser movimentada, pois baixa coesão é um indício que a classe está "mal posicionada" na estrutura do sistema de software e que a classe não se assemelha às demais classes do pacote. Isso deteriora a modularização do software, visto que a existência de classes com baixa coesão aumenta a responsabilidade do pacote, pois a classe fornece funções diferentes das classes presentes no pacote no qual ela está inserida. A HDI pode ser expressa por:

$$\text{HDI: } \{C_i \in P_i \wedge \text{Coesão}_{C_i} \leq \forall \text{Coesão}_{C_j} \in P_i\}$$

sendo P_i o pacote analisado e C_i as classes de P_i e Coesão_{C_i} o valor da medida TCC_p para a classe analisada;

b) Heurística de Decisão II (HDII) - Escolher classe com maior similaridade funcional.

Nesse contexto, similaridade funcional é o quanto a funcionalidade da classe está relacionada com a funcionalidade do pacote destino. Portanto, quanto maior a similaridade funcional da classe, mais compatível é a sua funcionalidade com a funcionalidade do pacote destino e maior 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 a quantidade de classes desse pacote:

$$SimFunc(C_{P_i}) = \frac{|Dep(c)|}{|P_{Dest(c)}|}, \forall\, P_{Dest(c)}$$

sendo $SimFunc(C_{P_i})$ a similaridade funcional da classe C_i com o pacote destino (P_{Dest}), $Dep(C)$ a dependência da classe C com as classes do pacote destino e $|P_{Dest(c)}|$ a quantidade de classes do pacote P para onde a classe será movimentada. A HDII pode ser expressa por:

$$\text{HDII: } \{C \,/\, \text{MAX}(\text{SimFunc}(C_{P_i}), \forall\, C_{HDI})\}$$

sendo C_{HDI} o conjunto de classes resultante da utilização da Heurística de Decisão I;

c) Heurística de Decisão III (HDIII) - Escolher classe com maior Instabilidade.

Nesse contexto, Instabilidade é o quanto uma entidade (método, classe ou pacote) pode ser impactada por causa das modificações efetuadas em outras partes do sistema (Martin, 1994). Seguindo essa definição, deve-se movimentar a classe com maior Instabilidade, pois essa é mais suscetível a sofrer impactos por causa de modificações em outras partes do sistema de software. Sendo assim, ao priorizar o movimento de classes com maior Instabilidade, busca-se um sistema de software mais estável:

$$Instabilidade(C_i) = \frac{Ce_i}{|Ca_i + Ce_i|}$$

sendo Ce_i e Ca_i o valor das medidas Ce e Ca para a classe i, respectivamente. A HDIII pode ser expressa por:

$$\text{HDIII: } \{C \,/\, \text{MAX}(\text{Instabilidade}(C_i), \forall\, C_{HDII})\}$$

sendo C_{HDII} o conjunto de classes resultante da utilização da Heurística de Decisão II;

d) Heurística de Decisão IV (HDIV) - Escolher classe que gere menor impacto.

Nesse contexto, impacto é a quantidade de classes que devem ter o *status* das suas dependências atualizadas[6] em decorrência da classe movimentada, para referenciar adequadamente essa classe e não alterar o funcionamento do sistema de software. Portanto, quanto menor a quantidade de classes impactadas, menor a quantidade de atualizações em dependências necessárias para manter o funcionamento do sistema de software. A HDIV pode ser expressa por:

$$\text{HDIV: } \{C \,/\, \text{MIN}(\text{QuantDependência}(C_i), \forall\, C_{HDIII})\}$$

sendo QuantDependência(C_i) a quantidade de dependências da classe i e C_{HDIII} o conjunto de classes resultante da utilização da Heurística de Decisão III;

e) Heurística de Decisão V (HDV) - Escolher classe com a maior acoplamento.

Com essa heurística, a classe com maior acoplamento é selecionada para ser movimentada, pois alto acoplamento com classes externas ao pacote da classe analisada é um indício que a classe está "mal posicionada" na estrutura do sistema de software. Esse alto acoplamento é prejudicial, pois uma classe altamente acoplada é mais susceptível a sofrer impactos por modificações em outras partes do sistema de software. A HDV pode ser expressa por:

$$\text{HDV: } \{C \,/\, \text{MAX}\{\text{Acoplamento}_{C_i}\}, \forall\, C_{HDIV}\}$$

sendo P_i o pacote analisado, C_i as classes de P_i e C_{HDIV} o conjunto de classes resultante da utilização da Heurística de Decisão IV.

Após a utilização dessas heurísticas, se ainda existirem duas ou mais classe com mesma probabilidade movimentação (PMov), então a classe a ser movimentada é escolhida aleatoriamente do conjunto de classes resultantes da HDV, tendo em vista que a escolha de uma ou de outra classe não terá influência significativa no resultado.

[6] Atualizar as classes dependentes da classe movimentada, modificando o pacote em que essa se encontra para refletir seu posicionamento na estrutura de pacotes do sistema após sua movimentação.

6.4.3 Heurísticas de convergência

As heurísticas do grupo de convergência têm como objetivo determinar se a solução atual converge para a solução sugerida. Essas heurísticas são utilizadas no processamento da heurística SA para avaliar se uma solução é "melhor" que outra considerando os atributos acoplamento e coesão. Esse grupo é composto por quatro heurísticas:

a) `Heurística de Convergência I (HCI) - Aceitar solução sugerida se o valor de acoplamento e de coesão melhorou.`
   Nesse caso, aceita-se a solução sugerida, pois a análise do valor do deltaAcoplamento e do deltaCoesão indica que essa solução apresenta melhor valor do acoplamento e da coesão que a solução atual. A `HCI` pode ser expressa por:

   $$\text{HCI: se deltaAcoplamento} \leq 0 \wedge \text{deltaCoesão} \geq 0 \text{ então aceitarSolução}$$

b) `Heurística de Convergência II (HCII) - Aceitar solução sugerida se o valor de acoplamento melhorou e o valor da coesão piorou, desde que a melhora no acoplamento seja melhor que a piora na coesão.`
   Nesse caso, a aceitação da nova solução é condicional, sendo necessário verificar se o ganho no valor do acoplamento é superior à perda no valor da coesão, por meio da análise do valor do deltaAcoplamento e do deltaCoesão. A `HCII` pode ser expressa por:

   $$\text{HCII: se deltaAcoplamento} \leq 0 \wedge \text{deltaCoesão} < 0 \wedge \text{FuncObjAcoplamento} > \text{FuncObjCoesão então aceitarSolução}$$

c) `Heurística de Convergência III (HCIII) - Aceitar solução sugerida se o valor do acoplamento piorou e a coesão melhorou, desde que a melhora na coesão seja melhor que a piora no acoplamento.`
   Nesse caso, a aceitação da nova solução é condicional, sendo necessário verificar se o ganho no valor da coesão é superior à perda no valor do acoplamento, por meio da análise do valor do deltaAcoplamento e do deltaCoesão. A `HCIII` pode ser expressa por:

   $$\text{HCIII: se deltaAcoplamento} > 0 \wedge \text{deltaCoesão} \geq 0 \wedge \text{FuncObjCoesão} >$$

FuncObjAcoplamento então aceitarSolução

d) Heurística de Convergência IV (HCIV) - Aceitar solução sugerida se o valor da coesão e do acoplamento pioraram e a heurística *Simulated Annealing* aceita essa solução.

Nesse caso, o acoplamento e a coesão do sistema de software foram deteriorados pelo movimento da classe entre pacotes. Sob o ponto de vista "lógico", deve-se rejeitar a solução sugerida, visto que essa solução possui valores piores do acoplamento e da coesão, detectados por meio da análise do valor do deltaAcoplamento e do deltaCoesão. Entretanto, se a heurística SA "aceita" essa solução para evitar convergência para mínimos locais no espaço de busca, então deve-se aceitar a solução sugerida. Os critérios para a SA aceitar uma solução "pior" são descritos na Seção 5.3. Por isso, utilizando a SA, existe a possibilidade de aceitar um movimento que deteriore a estrutura atual, mas, por consequência, acarreta a execução de sucessivos movimentos que a aprimoram, obtendo uma solução melhor ao final. A HCIV pode ser expressa por:

$$\text{HCIV: se deltaAcoplamento} > 0 \wedge \text{deltaCoesão} < 0 \wedge \text{SA} = \text{aceitaSolução então aceitarSolução}$$

sendo SA a avaliação da solução sugerida pela heurística SA para verificar se, na iteração atual, essa heurística aceita a deterioração em busca de uma solução melhor no espaço de busca e para evitar mínimos locais.

6.4.4 Descrição da abordagem para reestruturação de sistemas de software

ARS (**A**bordagem para **R**eestruturação de **S**oftware) é composta por dez atividades (Figura 6.2). A primeira atividade consiste em determinar a porcentagem máxima de deterioração (P) aceitável em um atributo (*e.g.*, coesão) em prol do aprimoramento proporcionado ao outro atributo analisado (*e.g.* acoplamento) (**Definir porcentagem de deterioração** - Figura 6.2). Essa atividade é realizada pelo usuário para determinar os patamares que a solução retornada por ARS deve atender para satisfazer suas necessidades.

Em seguida, a medição dos atributos acoplamento e coesão do sistema de software com as medidas de software selecionadas é iniciada (**Medir software** - Figura 6.2). Essa medição é realizada utilizando as medidas apresentadas no Capítulo 4 e seguindo as adequações de granularidade apresentadas na Seção 6.2. Com base no valor das medidas do grupo de

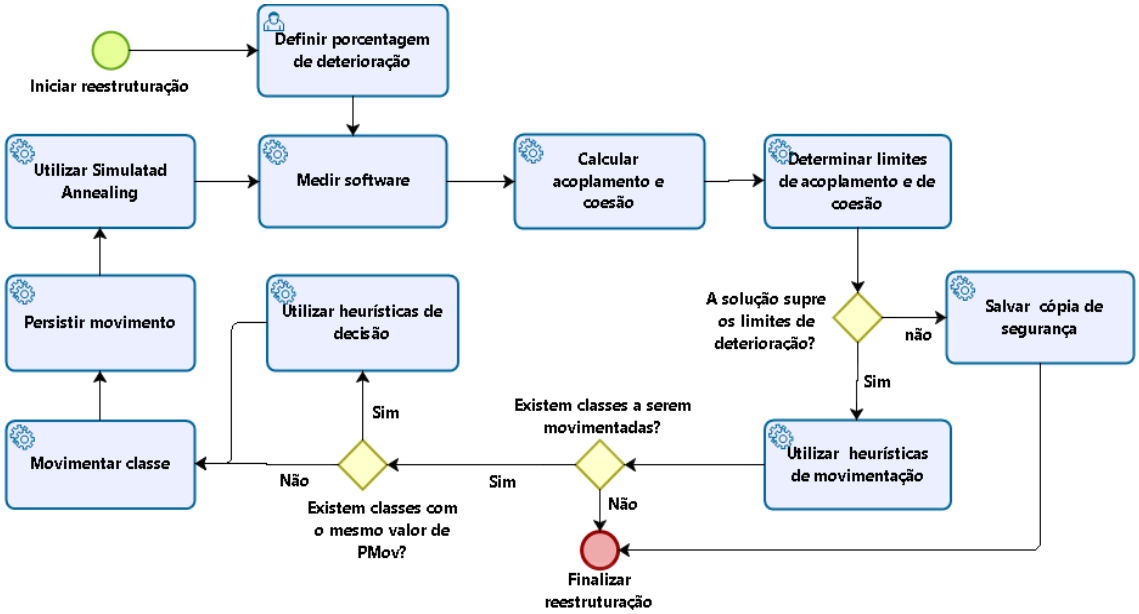
reestruturação, é calculado o estado atual dos atributos acoplamento e coesão (**Calcular acoplamento e coesão** - Figura 6.2). Para obter o valor desses atributos, foram utilizados:

$$acoplamento = CBO_p + RFC_p + MPC_p$$

$$coesão = TCC_p$$

Para obter o valor do acoplamento, foi utilizada a soma do valor das medidas CBO_p, RFC_p e MPC_p por aferir o mesmo atributo sob pontos de vista diferentes. Além disso, essas medidas possuem mesmo sentido de crescimento, não possuem valores inversos e apresentam correlação entre elas (BARBOSA; HIRAMA, 2013; LIMA; RESENDE, 2014; SILVA; COSTA, 2015). Logo, se o valor de uma medida aumentar ou reduzir, o valor das demais tende a seguir o mesmo comportamento, mesmo que em escalas diferentes. Dessa maneira, pela observação da soma desses valores, pode ser constatada melhoria ou piora na estrutura do sistema de software analisado. Sendo assim, quanto maior o valor expresso pelo acoplamento, pior é a estrutura do sistema de software.

Figura 6.2 - Visão Geral de ARS



Fonte: Do autor (2017).

Finalizada a medição, os limites de acoplamento e de coesão é determinado com base nas porcentagens máximas de deterioração definidas pelo usuário (**Determinar limites de acoplamento e de coesão** - Figura 6.2). O objetivo desses limites é determinar a necessidade de armazenar a última solução obtida que atenda os limites definidos para garantir que, ao final da reestruturação, a "melhor" solução encontrada nos limites de deterioração seja retornada ao usuário. O armazenamento dessa solução justifica-se pela necessidade de avaliar as soluções

obtidas pelo movimento de uma classe, pois, mesmo que a solução atual exceda os limites definidos, ela pode levar a uma solução que alcance os objetivos pretendidos de forma melhor do que a melhor solução encontrada até o momento.

Para obter o valor dos limites de deterioração, é calculado o produto entre a porcentagem de deterioração aceitável referente ao valor do acoplamento ou ao valor da coesão, definidos previamente pelo usuário, e o valor do seu respectivo atributo na estrutura do sistema de software. Para obter esses valores, foram utilizados:

$$\textit{Limite de deterioração do acoplamento} = \textit{acoplamento} * P$$

$$\textit{Limite de deterioração da coesão} = \textit{coesão} * P$$

Por exemplo, caso o usuário defina que a porcentagem de deterioração para o valor do acoplamento seja `30%`, a porcentagem de deterioração para o valor da coesão seja `20%` e os valores iniciais desses atributos sejam, respectivamente, `100` e `100`, os limites a serem respeitados são, respectivamente, `130` e `80`. Dessa forma, se algum dos limites definidos for violado, uma cópia da última solução que os atenda é armazenada e a reestruturação segue seu fluxo normal. Caso, posteriormente, uma solução melhor seja encontrada, essa assume o lugar da solução armazenada, garantindo que a "melhor" solução obtida no espaço de busca seja retornada para o usuário.

Determinados os limites de deterioração, é verificado se esses limites são respeitados pela estrutura atual do software (**Solução supre os limites de deterioração?** - Figura 6.2). Caso um dos limites seja violado, uma cópia da última estrutura que os satisfaça é armazenada e a reestruturação segue seu fluxo normal (**Salvar cópia de segurança** - Figura 6.2). O processo de reestruturação poderia ser finalizado nesse ponto, porém isso não é feito, pois o movimento realizado, que deteriorou a estrutura do sistema de software violou os limites de deterioração, pode desencadear movimentos que a aprimoram, gerando uma estrutura melhor que a armazenada em iterações anteriores. Se isso acontecer, a cópia armazenada é descartada.

Em seguida, as heurísticas de movimentação são utilizadas para determinar a probabilidade de cada classe ser movimentada entre os pacotes do sistema de software (**Utilizar heurísticas de movimentação** - Figura 6.2). Com base nessas probabilidades, é verificado se existem classes que podem ser movimentadas entre pacotes (**Existem classes a serem movimentadas?** - Figura 6.2). Para isso, a estrutura do software é percorrida a procura da classe com o maior valor de `PMov` para essa classe ser movimentada. Se não existirem classes a serem movimentadas (`PMov` $\leq$ `0`), então a estrutura atual é sugerida ao usuário, pois as classes estão

no pacote visto como o “mais adequado” pela ARS. Caso contrário, procura-se a classe com o maior valor de `PMov` dentre as classes do sistema de software analisado. Também, é verificada a existência de classes com o mesmo valor de `PMov` (**Existem classes com o mesmo valor de *PMov*? -** Figura 6.2). Em caso positivo, as heurísticas de decisão são utilizadas para escolher a “melhor” classe a ser movimentada (**Utilizar heurísticas de decisão** - Figura 6.2).

Posteriormente, a classe escolhida é movimentada para o seu pacote destino (**Movimentar classe** - Figura 6.2), gerando uma nova estrutura para o sistema (solução sugerida). Nesse ponto, o movimento realizado é persistido na estrutura da solução sugerida, atualizando o *status* das dependências das classes que se relacionam com a classe movimentada e o valor das medidas dessa estrutura (**Persistir movimento** - Figura 6.2). Isso tem como objetivo manter o comportamento do sistema inalterado e concretizar a reestruturação.

Com base na solução sugerida por ARS, SA verifica se a estrutura da solução sugerida é melhor que a estrutura da solução atual (**Utilizar *Simulated Annealing*** - Figura 6.2). Com isso, a melhor entre essas soluções é a base para gerar uma nova solução para tentar melhorar a estrutura do sistema de software. Esse processamento de ARS (Figura 6.2) repete-se até a inexistência de classes com probabilidade positiva de serem movimentadas (`PMov > 0`) para gerar uma nova solução.

Com essa explicação de ARS, o seu relacionamento com a heurística SA torna-se perceptível, pois essa heurística utiliza ARS para gerar a solução vizinha, possibilitando o processamento dela e a realização da reestruturação. Uma das vantagens adicionada pela utilização de SA refere-se a evitar a subjetividade da avaliação humana e os mínimos locais no espaço de busca.

6.5 Exemplo de aplicação da abordagem para reestruturação de sistemas de software

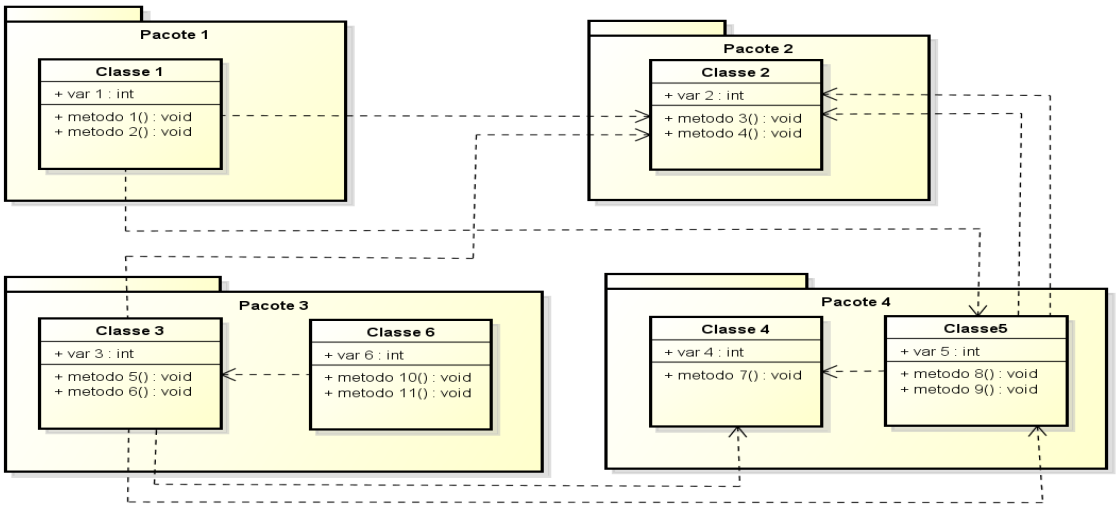
Para melhor compreender ARS, um exemplo de uso é apresentado utilizando um sistema de software hipotético representado pelo Diagrama de Pacotes da UML (Figura 6.3). As dependências existentes entre as classes nesse exemplo são especificadas no Diagrama de Sequência apresentado na Figura 6.4.

O primeiro passo na execução da abordagem é a definição das porcentagens máximas de deterioração (`P`) relacionadas aos atributos acoplamento e coesão. Nesse exemplo, é utilizado `50%` para ambos os atributos (objetivando facilitar os cálculos). Em seguida, é iniciada a medição do sistema de software, utilizando as medidas escolhidas, cujas definição original e

exemplificação são apresentadas no Capítulo 4 e suas alterações de granularidade são descritas na Seção 6.2, em que as medidas são calculadas para as classes na granularidade de pacotes, considerando somente relacionamentos interpacotes. Depois, é apresentado um exemplo do cálculo de cada medida utilizada na estrutura do sistema de software exemplo:

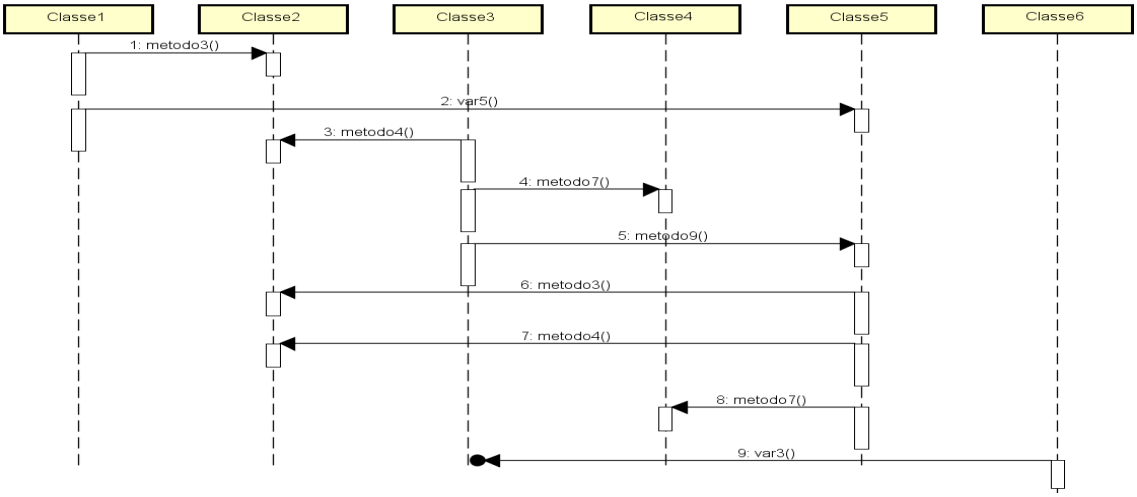
a) **MPC_p** - O valor dessa medida para as classes do exemplo da Figura 6.3 é apresentado na Tabela 6.1. Por exemplo, o valor da medida MPC_p das classes `Classe2`, `Classe4` e `Classe6` é igual a `0`, pois essas classes não chamam métodos fora do seu pacote. Por outro lado, a classe `Classe1` tem o valor da medida MPC_p igual a `1`, pois ela chama um método da classe `Classe2`;

Figura 6.3 - Diagrama de Pacotes do Sistema de Software Exemplo



Fonte: Do autor (2017).

Figura 6.4 - Diagrama de Sequência do Sistema de Software Exemplo



Fonte: Do autor (2017).

Tabela 6.1 - Cálculo da Medida MPC_p

Classes	Conjunto de Métodos Chamados	Valor de MPC_p
Classe1	{metodo3}	1
Classe2	{ }	0
Classe3	{metodo3; metodo7; metodo9}	3
Classe4	{ }	0
Classe5	{metodo3; metodo4}	2
Classe6	{ }	0

Fonte: Do autor (2017).

b) **CBO_p** - O valor dessa medida para as classes do exemplo da Figura 6.3 é apresentado na Tabela 6.2. Por exemplo, o valor da medida CBO_p para as classes Classe2, Classe4 e Classe6 é igual a 0, pois essas classes não estão acopladas a classes externas aos seus pacotes. Por outro lado, o valor da medida CBO_p da classe Classe3 é 3, pois essa classe está acoplada a três classes externas ao seu pacote (Classe2, Classe4 e Classe5);

Tabela 6.2 - Cálculo da Medida CBO_p

Classes	Conjunto de Classes Acopladas	Valor de CBO_p
Classe1	{Classe2; Classe5}	2
Classe2	{ }	0
Classe3	{Classe2; Classe4; Classe5}	3
Classe4	{ }	0
Classe5	{Classe2}	1
Classe6	{ }	0

Fonte: Do autor (2017).

c) **RFC_p** - O valor dessa medida para as classes do exemplo da Figura 6.3 é apresentado na Tabela 6.3. Por exemplo, o valor da medida RFC_p para a classe Classe1 é igual a 3, pois essa classe possui dois métodos e chama mais um método externo ao pacote. Por outro lado, o valor da medida RFC_p da classe Classe2 é igual 2, pois ela possui dois métodos e não chama métodos pertencentes a classes externas ao seu pacote;

Tabela 6.3 - Cálculo da Medida RFC_p

Classes	Quant. de Métodos na Classe	Conj. de Métodos Chamados	Valor de RFC_p
Classe1	2	{metodo3}	3
Classe2	2	{ }	2
Classe3	2	{metodo3; metodo7; metodo9}	5
Classe4	1	{ }	1
Classe5	2	{metodo3; metodo4}	4
Classe6	2	{ }	2

Fonte: Do autor (2017).

d) **Ca** - O valor dessa medida para o exemplo da Figura 6.3 é apresentado na Tabela 6.4. Por exemplo, o valor da medida Ca para as classes `Classe1`, `Classe3` e `Classe6` é igual a `0`, pois essas classes não possuem dependências incidentes sobre elas vindas de classes externas aos seus pacotes. Por outro lado, a classe `Classe5` tem o valor da medida Ca igual a `2`, pois as classes `Classe1` e `Classe3` são suas dependentes;

Tabela 6.4 - Cálculo da Medida Ca

Classes	Conjunto de Classes que Dependem da Classe Analisada	Valor de Ca
Classe1	{ }	0
Classe2	{Classe1; Classe3; Classe5}	3
Classe3	{ }	0
Classe4	{Classe3}	1
Classe5	{Classe1; Classe3}	2
Classe6	{ }	0

Fonte: Do autor (2017).

e) **Ce** - O valor dessa medida para as classes do exemplo da Figura 6.3 é apresentado na Tabela 6.5. Por exemplo, o valor da medida Ce para as classes `Classe2`, `Classe4` e `Classe6` é igual a `0`, pois essas classes não dependem de classes externas aos seus pacotes. Por outro lado, o valor da medida Ce para a classe `Classe3` é igual a `3`, pois essa classe depende das classes `Classe2`, `Classe4` e `Classe5`;

Tabela 6.5 - Cálculo da Medida Ce

Classes	Conjunto de Classes que a Classe Analisada Depende	Valor de Ce
Classe1	{Classe2; Classe5}	2
Classe2	{ }	0
Classe3	{Classe2; Classe4; Classe5}	3
Classe4	{ }	0
Classe5	{Classe2}	1
Classe6	{ }	0

Fonte: Do autor (2017).

f) **$LCOM4_p$** - O valor dessa medida para os pacotes do exemplo da Figura 6.3 é apresentado na Tabela 6.6. Por exemplo, o valor da medida $LCOM4_p$ para os pacotes do sistema de software analisado é igual a `1`, pois, em todos esses pacotes, as classes estão totalmente conectadas, formando um único grafo conexo em cada pacote. Essa medida, ao contrário das demais, é calculada para cada pacote, pois deseja-se saber a coesão do pacote analisado e não da classe, que não se altera com a movimentação de classes entre pacotes;

g) **TCC_p** - O valor dessa medida para as classes do exemplo da Figura 6.3 é apresentado na Tabela 6.7. Por exemplo, o valor da medida TCC_p para a classe `Classe3` é igual a `1`, pois, aplicando as fórmulas descritas na Seção 6.2, seu NDC é igual a `1` (quantidade de métodos que utilizam a classe) e seu NP é igual a `1` $\left(NP = \frac{2*(2-2)}{2} = 1\right)$ resultando em um valor para a medida TCC_p igual a `1` $\left(TCC_p(C) = \frac{1}{1} = 1\right)$;

Tabela 6.6 - Cálculo da Medida $LCOM_p$

Pacote	Classes que Compõem o Pacote	Vértices dos Grafos Construídos	Quantidade de Grafos Obtidos	Valor de $LCOM4_p$
Pacote1	{Classe1}	{Classe1}	1	1
Pacote2	{Classe2}	{Classe2}	1	1
Pacote3	{Classe3; Classe6}	{Classe3; Classe6}	1	1
Pacote4	{Classe4; Classe5}	{Classe4; Classe5}	1	1

Fonte: Do autor (2017).

Tabela 6.7 - Cálculo da Medida TCCp

Classe	Conjunto de Métodos que Utilizam a Classe Analisada	Quantidade de Métodos da Classe	NDC	NP	Valor de TCC_p
Classe1	{ }	2	0	1	0
Classe2	{ }	2	0	1	0
Classe3	{metodo10}	2	1	1	1
Classe4	{metodo7}	1	1	0	0
Classe5	{ }	2	0	1	0
Classe6	{ }	2	0	1	0

Fonte: Do autor (2017).

Obtidos o valor das medidas para a classe ou pacote, o valor dessas medidas para o sistema de software é calculado. Para isso, é somado o valor das medidas das classes que compõem o sistema ou somado o valor da medida dos pacotes que compõem o sistema (medida $LCOM4_p$). A soma desses valores é apresentada na Tabela 6.8. Por exemplo, o valor da medida RFC_p para o sistema de software exemplo é igual a `17` (`3 + 2 + 5 + 1 + 4 + 2` - Tabela 6.3).

Com base nos valores dessas medidas para o sistema de software e aplicando as fórmulas apresentadas na Seção 6.2, são determinados o valor dos atributos acoplamento e coesão do sistema de software exemplo, respectivamente, `29` e `1`:

$$acoplamento_{sistema} = CBO_p + RFC_p + MPC_p = 6 + 17 + 6 = 29$$

$$coesão_{sistema} = TCC_p = 1$$

Em seguida, são determinados os limites de acoplamento e de coesão, sendo `43,5` (`50%` de aumento em relação ao valor de acoplamento atual do software) e `0,5` (`50%` de redução em relação ao valor de coesão atual do software), respectivamente. Seguindo os passos de ARS, é verificado se o estado atual do sistema de software respeita os limites definidos. Realizando essa verificação, é constatado que o valor do acoplamento da estrutura atual (`29`) respeita o limite de acoplamento definido (`43,5`), pois seu valor é inferior ao limite definido e o valor da coesão da estrutura atual (`1`) respeita o objetivo de coesão (`0,5`), pois seu valor é superior ao limite de coesão. Portanto, a reestruturação pode prosseguir.

Tabela 6.8 - Valores das Medidas para o Sistema de Software

Medidas	Valores para o Sistema de Software
MPC_p	6
CBO_p	6
RFC_p	17
Ca	6
Ce	6
$LCOM4_p$	4
TCC_p	1

Fonte: Do autor (2017).

Continuando a reestruturação, as heurísticas de movimentação são utilizadas, determinando a probabilidade de cada classe ser movimentada entre os pacotes. Ao utilizar `HMI` no exemplo da Figura 6.3, o valor `0` (zero) é atribuído à probabilidade de movimentação (`PMov`) para as classes `Classe1` e `Classe2`, evitando a sua movimentação. Em seguida, `HMII` é utilizada para determinar a probabilidade de movimentação das demais classes do sistema de software. O valor de `PMov` resultante da utilização dessas heurísticas para cada classe em relação seus respectivos pacotes vizinhos é representada na Tabela 6.9. Esse valor para algumas classes é igual a `0` em relação aos pacotes vizinhos, pois elas estão sozinhas em seus pacotes origens. Para outras classes, a informação apresentada é "–" indicando que o pacote é onde a classe se encontra ou o pacote listado na coluna não é um pacote vizinho dessa classe (não possui relacionamentos com a classe). As classes `Classe3` e `Classe5` possuem valor de `PMov` igual a `1` com os pacotes `Pacotes4` e `Pacote2`, respectivamente, indicando que elas são candidatas para movimentação para os respectivos pacotes.

Definidas as probabilidades de movimentação, a estrutura do sistema é percorrida buscando a classe com maior valor de `PMov` para ser movimentada. No exemplo da Figura 6.3, as classes `Classe3` e `Classe5` possuem a mesma `PMov` (`1`); por isso, para definir qual dessas classes deve ser movimentada, são utilizadas as heurísticas de decisão.

Tabela 6.9 - Probabilidade de Movimentação das Classes do Sistema de Software

Classes	Pacote1	Pacote2	Pacote3	Pacote4
Classe1	-	0	-	0
Classe2	0	-	0	0
Classe3	-	0	-	1
Classe4	-	-	0	-
Classe5	0	1	0	-
Classe6	-	-	-	-

Fonte: Do autor (2017). Legenda: "-" indica que a classe não se relaciona com o respectivo pacote.

Utilizando a HDI, a classe com a menor coesão é escolhida para ser movimentada. Nesse exemplo, a classe Classe3 possui coesão igual a 1 e a classe Classe5 possui coesão igual a 0. Assim, a classe Classe5 é eleita para ser movimentada, com o objetivo de proporcionar maior ganho de coesão ao sistema de software reestruturado. Visto que não foram encontradas outras classes com o mesmo valor de coesão, as demais heurísticas de decisão não serão utilizadas, pois uma única classe foi encontrada para ser movimentada.

O próximo passo é movimentar a classe escolhida do pacote origem para o pacote destino. Nesse exemplo, a classe Classe5 é movimentada do pacote Pacote5 (origem) para o pacote Pacote2 (destino) (Figura 6.5). Em seguida, o valor das medidas utilizadas é recalculado para obter o estado atual da estrutura sugerida. Assim, os valores são atualizados para as classes (Tabela 6.10) ou para o pacote no caso de $LCOM4_p$ (Tabela 6.11) do sistema de software. Por exemplo, o valor da medida Ca para a classe Classe2 era 3 antes da movimentação da classe Classe5 (Tabela 6.4) e após o movimento o valor dessa medida tornou-se 2.

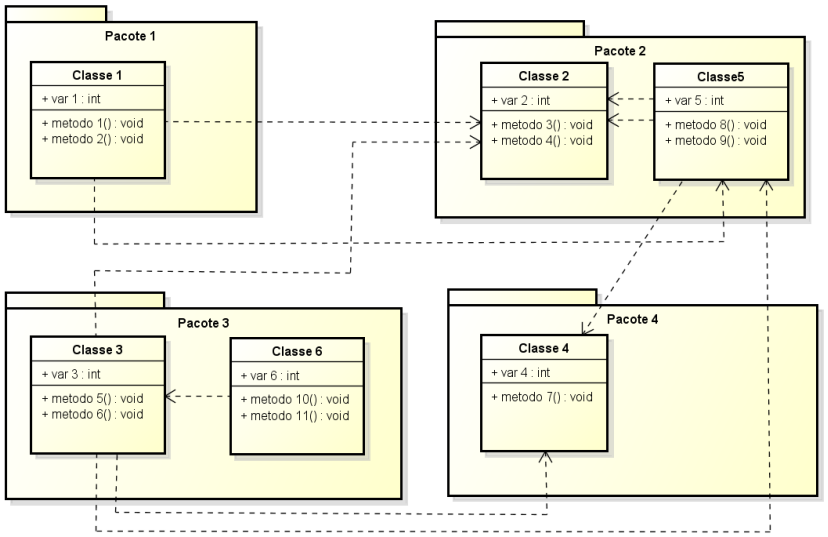
Em seguida, na Tabela 6.12, os valores de cada classe ou pacote são somados para obter o valor das medidas para o sistema de software. Por exemplo, valor da medida Ca após o movimento da classe Classe5 é igual a 6, mesmo valor dessa medida antes do movimento. Por outro lado, o valor da medida RFC_p com o movimento da classe Classe5 passou de 17 para 14, reduzindo o valor dessa medida de acoplamento (Tabela 6.8).

Atualizado o valor das medidas de software utilizadas, o valor dos atributos acoplamento e coesão da estrutura atual do sistema de software são atualizados. Nesse exemplo, 25 e 3, respectivamente:

$$acoplamento_{sistema} = CBO_p + RFC_p + MPC_p = 6 + 14 + 5 = 25$$

$$coesão_{sistema} = TCC_p = 3$$

Figura 6.5 - Diagrama de Pacotes do Sistema de Software Após Movimentar a Classe Classe5 do Pacote Pacote5 para o Pacote Pacote2



Fonte: Do autor (2017).

Tabela 6.10 - Valores Atualizados das Medidas para as Classes do Sistema de Software

Medidas	Classe1	Classe2	Classe3	Classe4	Classe5	Classe6
Ca	0	2	0	2	2	0
Ce	2	0	3	0	1	0
CBO_p	2	0	3	0	1	0
MPC_p	1	0	3	0	1	0
RFC_p	3	2	5	1	1	2
TCC_p	0	2	1	0	0	0

Fonte: Do autor (2017).

Tabela 6.11 - Valores Atualizados de LCOM4p para os Pacotes

Medidas	Pacote1	Pacote2	Pacote3	Pacote4
$LCOM4_p$	1	1	1	1

Fonte: Do autor (2017).

Nesse ponto, tem-se uma solução sugerida e a heurística SA verifica se essa solução deve ser aceita seguindo o seu processamento (Seção 5.3) e as adaptações (Seção 6.3). Utilizando a SA, os valores da função objetivo de cada solução em relação ao acoplamento e à coesão devem ser definidos. Calculando esses valores para a solução atual, que nesse caso refere-se à solução original, foi obtido `100%` para as duas funções, pois o valor da coesão (`1`) e o valor do acoplamento (`29`) da solução atual são os mesmos da solução original.

$$FuncObjAcoplamento = \frac{29 * 100}{29} = 100\%$$

$$FuncObjCoesão = \frac{1 * 100}{1} = 100\%$$

Tabela 6.12 - Valores Atualizados das Medidas Utilizadas para o Sistema de Software

Medidas	Sistema de Software
Ca	6
Ce	6
CBO_p	6
MPC_p	5
RFC_p	14
TCC_p	3
$LCOM_p$	4

Fonte: Do autor (2017).

Realizando o mesmo cálculo para a solução sugerida, os resultados foram `86,2%` e `300%` para a função objetivo de acoplamento e para a função objetivo de coesão, respectivamente.

$$FuncObjAcoplamento = \frac{25 * 100}{29} = 86{,}2\%$$

$$FuncObjCoesão = \frac{3 * 100}{1} = 300\%$$

Com base nesses valores, os valores de `deltaAcoplamento` e de `deltaCoesão` são calculados:

$$deltaAcoplamento = (86{,}2 - 100) = -13{,}8$$

$$deltaCoesão = (300 - 100) = 200$$

Para decidir pela aceitação da solução sugerida, são utilizadas as heurísticas de convergência. Ao utilizar `HCI`, o resultado indica a aceitação da solução sugerida, pois o valor de acoplamento reduziu e o valor de coesão aumentou (`se -13,8 ≤ 0 ∧ 200 ≥ 0 então aceitarSolução`). Portanto, a solução sugerida deve substituir a solução atual, pois o acoplamento e a coesão foram aprimorados simultaneamente.

Após a aceitação da solução sugerida, ARS é novamente iniciada para gerar uma nova solução e tentar aprimorar a estrutura do sistema de software. ARS é executada de maneira cíclica até não encontrar mais oportunidades de aprimoramento para obter novas estruturas para o sistema de software. Entretanto, para simplificar, nessa exemplificação, é apresentada

somente a primeira iteração de ARS, visto que as demais iterações executam os mesmos passos apresentados.

6.6 Considerações finais

Neste capítulo, foi apresentada uma abordagem para reestruturação de sistemas de software (ARS). Para cada passo de ARS, foram detalhadas suas atividades e as estratégias empregadas para alcançar seu objetivo, como alterações na granularidade das medidas, limites de deterioração, heurísticas de movimentação, de decisão e de convergência e adaptações na heurística *Simulated Annealing*.

Para facilitar a compreensão do funcionamento de ARS, foi apresentado um exemplo de sua utilização sobre um sistema de software hipotético. Nesse exemplo, foi exemplificada a medição de cada uma das medidas utilizadas, a alteração na granularidade de medição, a utilização das heurísticas de movimentação, de decisão e de convergência, as funções objetivo, os deltas e os demais detalhes empregados na execução da reestruturação.

7 SRT - *SOFTWARE RESTRUCTURING TOOL*

Nesse capitulo é apresentada uma visão geral da ferramenta que automatiza a abordagem proposta para reestrutura sistemas de software.

7.1 Considerações iniciais

Neste capítulo, é apresentada a ferramenta ***S**oftware **R**estructuring **T**ool* (SRT), um *plug-in* desenvolvido na linguagem de programação Java para a plataforma Eclipse IDE. Essa plataforma foi utilizada por ser considerada a mais popular de desenvolvimento de código aberto Java (RUBEL, 2006; SHELAJEV; MAPLE, 2016) e por possuir arquitetura estruturada sobre o conceito de *plug-ins* (MURPHY et al., 2006). SRT automatiza a abordagem para reestruturação de software (ARS) (SANTOS; JUNIOR; COSTA, 2016c) que, além de ampliar a probabilidade de utilização dessa abordagem em cenários reais de produção, possibilita verificar a eficácia de ARS.

SRT foi desenvolvida para analisar sistemas de software desenvolvidos na linguagem de programação Java, por ser a mais popular do mundo nos últimos 15 anos (CASS, 2016; TIOBE, 2016). Entretanto, ressalta-se que ARS pode ser abstraída para qualquer linguagem orientada a objetos, não se limitando a Java. As principais funções de SRT são:

a) Reestruturar automaticamente sistemas de software desenvolvidos em Java;
b) Possibilitar restrição de pacotes que não devem ser reestruturados em conjunto;
c) Apresentar *log* dos movimentos realizados pela reestruturação;
d) Apresentar visualização das dependências interpacotes e interclasses do sistema de software analisado.

O restante deste capítulo está organizado da seguinte maneira. Uma visão geral das tecnologias utilizadas para implementar SRT são apresentados na Seção 7.2. A arquitetura empregada nessa ferramenta é exibida na Seção 7.3. A estrutura utilizada para manipular os dados coletados dos sistemas de software e tornar o processo de reestruturação mais eficiente é descrita na Seção 7.4. Um exemplo real de utilização de SRT é apresentado na Seção7.5.

7.2 Tecnologias empregadas

A plataforma empregada para o desenvolvimento de SRT foi `Eclipse IDE 4.4.2`[7] (Luna), utilizando o *Java Development Kit* (JDK 8). A funcionalidade dessa plataforma foi acrescida por:

a) ***Plug-in* Development Environment**[8] **(PDE)** provê ferramentas que auxiliam o desenvolvedor nas fases de desenvolvimento, desde a criação até o empacotamento de um *plug-in*;

b) **Java Development Tools**[9] **(JDT)** possui um conjunto de *plug-ins* com diversos recursos para acessar e manipular código Java, permitindo acessar, criar e modificar projetos existentes no *workspace*. Os *plug-ins* que compõem o JDT são organizados em quatro categorias:
   - ***Annotation Processing Tool* (APT)** fornece *plug-ins* que compilam e reúnem informações sobre o código;
   - ***JDT Core*** provê acesso aos objetos do *Java Model* e manipuladores da estrutura Java presente nas ASTs (*Abstract Syntax Tree*);
   - ***JDT Debug*** provê *plug-ins* que fornecem suporte a execução e a depuração de código Java;
   - ***JDT UI*** implementa a interface de usuário para a plataforma Eclipse IDE, proporcionando contribuições para a visualização do *Workbench* e manipulação do código;

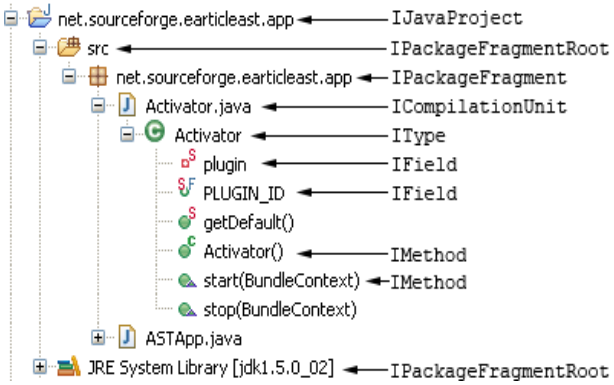
c) **Java Model** é um conjunto de classes que modelam objetos em memória (*Java elements*) que representam a estrutura do software Java existente no *Workspace* (Figura 7.1). Por meio da manipulação dos *Java elements*, são obtidas informações sobre o sistema de software e realizadas as movimentações das classes entre pacotes, pontos cruciais dessa abordagem de reestruturação;

d) **Abstract Syntax Tree (AST)** faz o mapeamento de cada elemento do código Java (1 - Figura 7.2) para um nó da árvore AST (2 - Figura 7.2), de forma semelhante a árvore DOM (*Document Object Model*) para o XML (*eXtensible Markup Language*). O responsável por realizar esse mapeamento é *ASTParser*, que transfora *strings* de código

[7] https://eclipse.org/luna/
[8] https://www.eclipse.org/pde/
[9] https://www.eclipse.org/jdt/

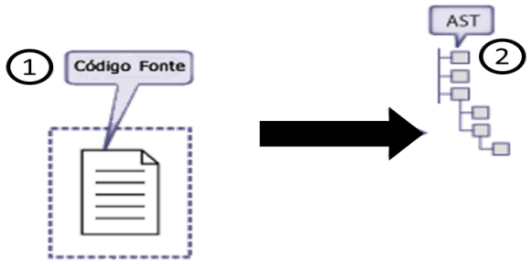
em nós de AST. Essa árvore pode ser construída a partir de qualquer *Java element*, tornando cada elemento filho um nó da árvore (*ASTNode*). Esses nós possuem um conjunto refinado de informações sobre o código, as quais não são possíveis de obter com o *Java Model*, porém o *Java Model* é mais rápido do que AST (Eclipse, 2015). Para obter informações dos nós da árvore, deve-se criar uma classe "visitante" que estenda *ASTVisitor* (classe abstrata base para visitar e obter dados de um *ASTNode*) para percorrer a árvore;

Figura 7.1 - *Java elements* Presentes no *Workspace*



Fonte: Do autor (2017).

Figura 7.2 - Mapeamento do Código para AST



Fonte: Do autor (2017).

e) **`JGraph`**[10] fornece funções que auxiliam na construção e no desenho de diagramas iterativos e grafos. Em SRT, `JGraph` foi utilizada para construir o diagrama de dependências entre pacotes e entre classes.

7.3 Arquitetura de SRT

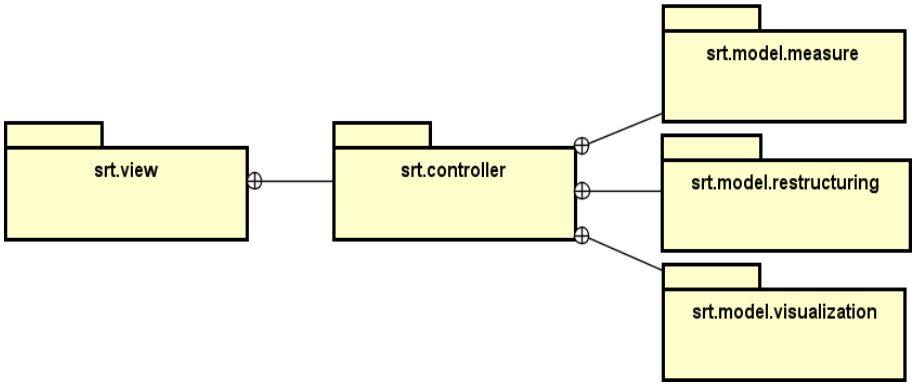
A estrutura de SRT está organizada em conformidade com o padrão arquitetural *Model View Controller* (MVC) (GAMMA et al., 1994) (Figura 7.3). Optou-se por esse padrão pois

[10] https://jgraph.github.io/mxgraph/

organiza o software em três camadas (modelo - *Model*, visão - *View* e controlador - *Controller*), realizando separação lógica entre as partes da aplicação, o que aumenta sua modularidade e facilita sua manutenção. Isso permite que partes diferentes sejam desenvolvidas paralelamente e possibilita modificar uma parte sem que as demais sofram impactos substanciais (GAMMA et al., 1994). Os pacotes que compõem essa arquitetura são:

a) Pacote `srt.view` apresenta ao usuário informações resultantes do processo de reestruturação (camada *View*);
b) Pacote `srt.model.measure` manipula as informações relacionadas à medição do sistema de software, utilizando JDT, *JavaModel* e AST para obter tais informações (camada *Model*);
c) Pacote `srt.model.restructuring` manipulada informações relacionadas à reestruturação, como a movimentação de classes entre pacotes, persistência dos movimentos realizados, a utilização dos grupos de heurísticas criados e a aplicação da heurística SA (camada *Model*);
d) Pacote `srt.model.visualization` manipula e constrói os diagramas de dependências interpacotes e interclasses apresentados na interface de SRT (camada *Model*);
e) Pacote `srt.controller` controla o fluxo de informações do sistema de software, determinando o que deve ser acionado em resposta às solicitações do usuário (camada *Controller*).

Figura 7.3 - Projeto Arquitetural do Apoio Computacional (*Plug-in*)



Fonte: Do autor (2017).

7.4 Estrutura de dados do software de SRT

Um dos pontos críticos na execução da reestruturação é obter a estrutura resultante de cada iteração de ARS, fato essencial para analisar se a estrutura obtida é melhor que a original. A cada iteração, é necessário realizar o movimento de uma classe entre pacotes e atualizar o *status* das dependências das classes que se relacionam com a classe movimentada, para manter o funcionamento do sistema de software inalterado. Além disso, essa atualização reflete diretamente sobre os valores das medidas de software que necessitam ser atualizados.

Uma forma de obter a estrutura resultante de ARS é, a cada iteração, movimentar a classe e persistir seu movimento diretamente sobre a estrutura do sistema de software existente no meio persistente, e, em seguida, remedir o software para obter os valores das medidas utilizadas, obtendo o estado atual desse sistema. Entretanto, realizar essa movimentação, atualização e medição diretamente no meio persistente, tornaria o processo "custoso", consumindo muitos recursos computacionais e tempo de processamento por causa do constante acesso ao meio persistente.

Para diminuir esse custo, foi elaborada e construída um tipo abstrato de dados (TAD) (Figura 7.4). Nesse TAD, em um objeto da classe `Software`, são armazenados o valor das medidas utilizadas, o valor dos estados original e atual dos atributos acoplamento e coesão, os limites a serem respeitados durante a reestruturação e as coleções de pacotes e de pacotes bloqueados, constituídas de objetos da classe `Pacote`. Em um objeto da classe `Pacote`, são armazenados o nome do pacote, o valor da medida $LCOM4_p$ e uma coleção de objetos da classe `Classe` no atributo `classes`. Em objetos da classe `Classe`, há atributos para armazenar o nome da classe, o nome do pacote que a contém, o valor das medidas utilizadas, a quantidade de métodos, o mapeamento contendo os métodos e/ou atributos que utilizam e são utilizados pela classe e o vizinho da classe `Vizinho`. Em um objeto da classe `Vizinho`, são armazenados o nome do pacote vizinho e a probabilidade que um objeto da classe `Classe` tem de ser movimentado para o pacote vizinho. Em um objeto da classe `Classe`, é armazenado, no máximo, um objeto da classe `Vizinho` no atributo `vizinho`, pois não é necessário armazenar todos os pacotes com os quais a classe se relaciona, mas somente o pacote que a classe possui maior probabilidade de ser movimentada. O TAD permite de forma rápida e prática realizar as seguintes operações necessárias para reestruturar o sistema de software:

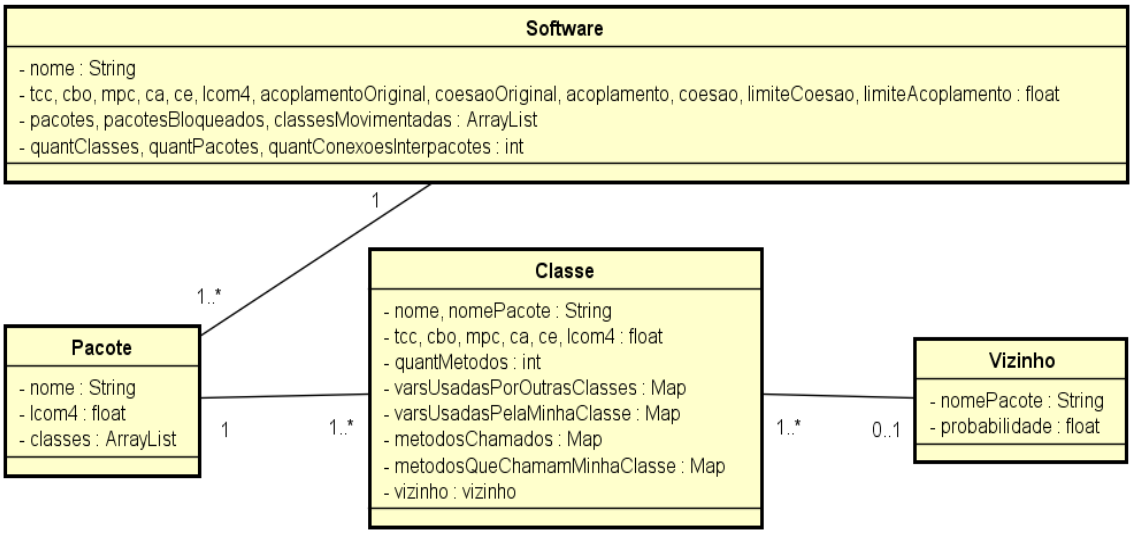
a) **Movimentar classes entre pacotes.** Para isso, é necessário movimentar o objeto da classe `Classe` entre as coleções de classes dos pacotes origem e destino;

b) **Persistir movimento realizado**. Atualizar *status* das classes que se relacionam com a classe movimentada. Para isso, basta percorrer as coleções das classes impactadas pelo movimento em busca da classe movimentada e atualizar o pacote em que essa se encontra;

c) **Recalcular medidas para o software em reestruturação**. O valor das medidas é recalculado somente para as classes impactadas e para a classe movimentada, com base nos atributos das classes, sem a necessidade de reanalisar o sistema de software existente em disco utilizando a JDT;

d) **Gerar diagrama de dependências**. Para isso, os elementos do sistema de software são "percorridos" para construir os vértices com base nos pacotes e, posteriormente, verificar as listas de dependências entre as classes para inserir as arestas e seus pesos.

Figura 7.4 - Tipo Abstrato de Dados para Armazenar Sistemas de Software em SRT



Fonte: Do autor (2017).

7.5 Exemplo de utilização de SRT

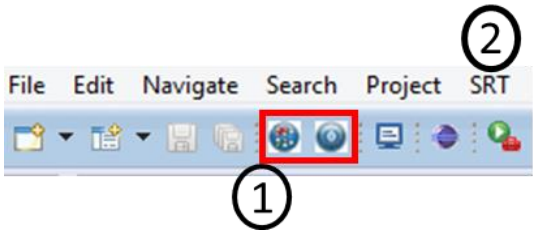
Nesta seção, é apresentada uma visão geral de utilização de SRT. Os requisitos que devem ser cumpridos para o sistema de software ser reestruturado por essa ferramenta são: i) estar presente no *workspace* do Eclipse; ii) disponibilizar acesso ao código fonte; e iii) ser desenvolvido em Java. Para ilustrar a utilização, foi utilizado o sistema de software `Playlist Sender`[11], que possui 51 classes organizadas em quatro pacotes.

[11] https://sourceforge.net/projects/playlister/?source=typ_redirect

Para iniciar SRT, deve-se selecionar um dos ícones da barra de ícones do *Eclipse* (**1** - Figura 7.5) ou a opção `SRT` da barra de *menu* (**2** - Figura 7.5). Na barra de ícones, o ícone a direita abre a tela na qual o usuário define o limite superior de deterioração do acoplamento e da coesão (Figura 7.6) e o ícone a esquerda abre a tela para iniciar a reestruturação (Figura 7.6). Na reestruturação, o sistema de software deve ser selecionado por meio do *combo box* `Select Software` (**1** - Figura 7.7), que lista os sistemas de software existentes no *workspace*, e sua medição é iniciada acionando o botão `OK` (**2** - Figura 7.7).

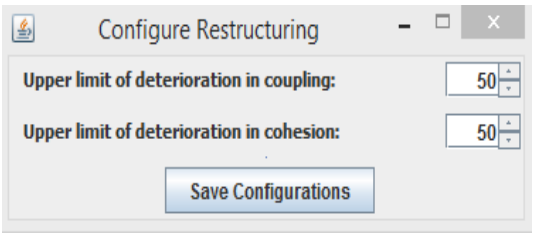
Após a medição, SRT apresenta o valor das medidas do Grupo de Avaliação e a visualização das dependências interpacotes e interclasses da estrutura original (**3** - Figura 7.7). Nesse ponto, o usuário pode acionar o botão `Restructuring` (**4** - Figura 7.7), em que SA e ARS interagem, executando o processamento descrito e exemplificado no Capitulo 6 para efetuar a reestruturação do sistema de software e sugerir uma estrutura "melhor" para esse sistema. Em seguida, a reestruturação é finalizada. Com isso, o valor das medidas do Grupo de Avaliação e a visualização das dependências interpacotes e interclasses da solução sugerida são apresentados na interface de SRT (**5** - Figura 7.7).

Figura 7.5 - Ícones de SRT na Barra de Ícones do Eclipse



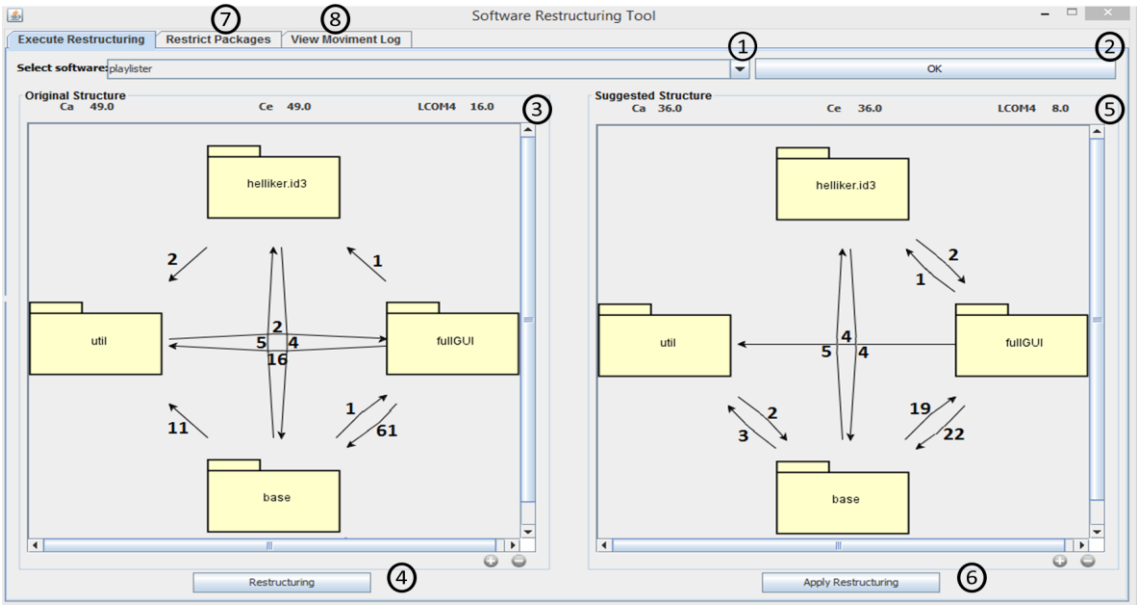
Fonte: Do autor (2017).

Figura 7.6 - Tela de Configuração dos Limites Superiores de Deterioração



Fonte: Do autor (2017).

Figura 7.7 - Tela de Execução da Reestruturação

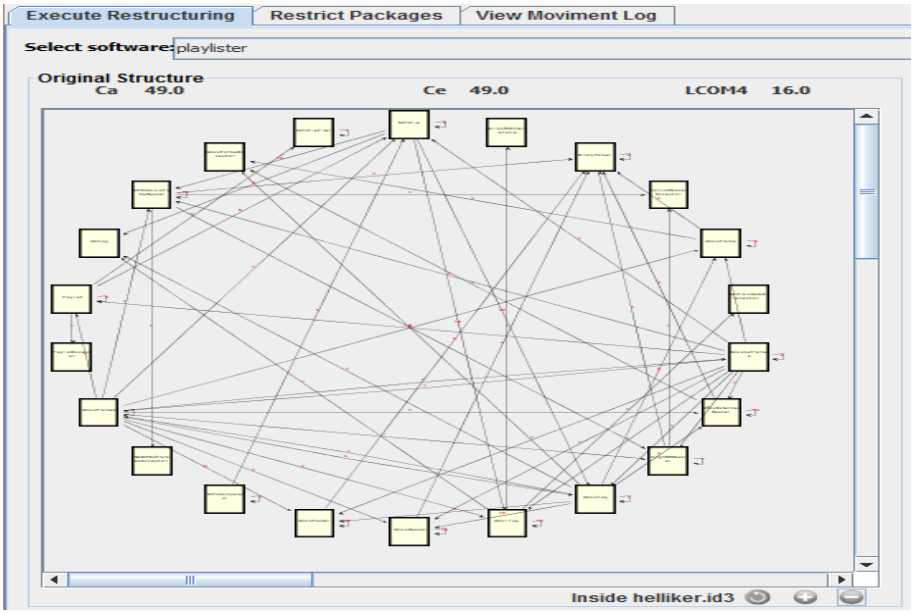


Fonte: Do autor (2017).

Na visualização do diagrama de dependências interpacotes, são apresentados os pacotes do sistema de software com as suas dependências indicadas por setas direcionais. Em cada uma dessas setas, ressalta-se a quantidade de dependências existentes entre os pacotes. Em SRT, o usuário pode aumentar ou reduzir a visualização, utilizando os botões + e −, respectivamente, localizados no canto inferior direito.

Além disso, o usuário pode acessar o diagrama de dependências interclasses ao “clicar” em um pacote. Por exemplo, ao clicar no pacote `helliker.id3` (Figura 7.7), o diagrama de dependências interclasses desse pacote, as setas direcionais para ressaltar as dependências entre as classes e um valor para indicar quantas vezes essa dependência ocorre são exibidos (Figura 7.8). O usuário pode aumentar ou reduzir a visualização, utilizando os botões + e −, respectivamente, localizados no canto inferior direito.

Figura 7.8 - Recorte da Tela para Visualização do Diagrama de Dependências Interclasses



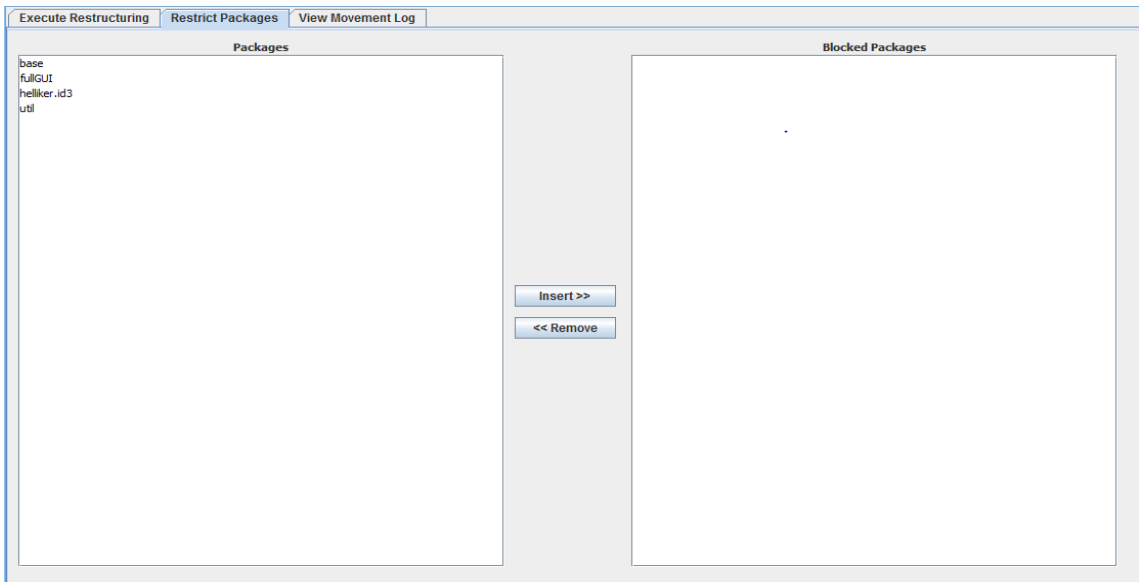
Fonte: Do autor (2017).

Observando a distribuição das dependências entre pacotes apresentadas pela solução original (**3 -** Figura 7.7) e pela solução sugerida (**5** -Figura 7.7), pode-se observar que as dependências entre pacotes foram aprimoradas. Por exemplo, entre os pacotes `Util` e `FullGUI`, houve redução de 16 dependências da solução original para 4 dependências na solução sugerida no sentido `FullGUI → Util`, e, no sentido oposto, as dependências existentes foram eliminadas. Em outras situações, as dependências se alteram de localização, não deixando de existir, como as dependências existentes entre os pacotes `Heilliker` e `Util` na solução original, excluídas na solução sugerida, mas passaram a existir dependências entre o pacote `Heilliker` e `FullGUI`; isso se deve à movimentação de classes entre os pacotes `Util` e `FullGUI`, que geraram o aprimoramento descrito anteriormente. A movimentação de classes também ocasiona o surgimento de dependências não existentes na solução original, como entre os pacotes `Base` e `Util`, fato ocasionado pela realocação de classes entre os pacotes do sistema de software.

Ao final da reestruturação, o usuário pode solicitar que a solução sugerida por SRT seja efetivada, acionando o botão `Apply Restructuging` (**6 -** Figura 7.7), fazendo com que as alterações estruturais executadas para a obtenção da solução sugerida sejam aplicadas (armazenadas em meio persistente). Portanto, o usuário é o responsável por tomar a decisão se a reestruturação foi satisfatória e deve ser armazenada. Sendo assim, SRT mantém a estrutura original do sistema de software intacta e gera uma cópia desse sistema com a estrutura interna alterada.

Além da reestruturação, há a possibilidade de o usuário limitar os pacotes a serem reestruturados em conjunto e respeitar a semântica dos pacotes. Isso pode ser feito por meio da aba `Restrict Packages` (**7** - Figura 7.7), em que o usuário pode inserir ou remover pacotes da lista de pacotes bloqueados, utilizando os botões `Insert` e `Remove` (Figura 7.9), respectivamente. Os pacotes presentes na lista de pacotes bloqueados não são considerados na reestruturação do sistema de software. As classes desses pacotes não são movimentadas nem classes de pacotes não bloqueados são movimentadas para dentro desses pacotes. Por exemplo, ao reestruturar um sistema de software construído sobre o padrão MVC, pode-se limitar a reestruturação somente aos pacotes da camada *Model*, inserindo os pacotes pertencentes à camada *View* e à camada *Controller* na lista de pacotes bloqueados. Dessa maneira, as classes pertencentes a camada *Model* não são enviadas para outras camadas, sendo remanejadas somente entre os pacotes dessa camada. Assim, os princípios desse padrão não são violados e a semântica existente entre os pacotes é respeitada.

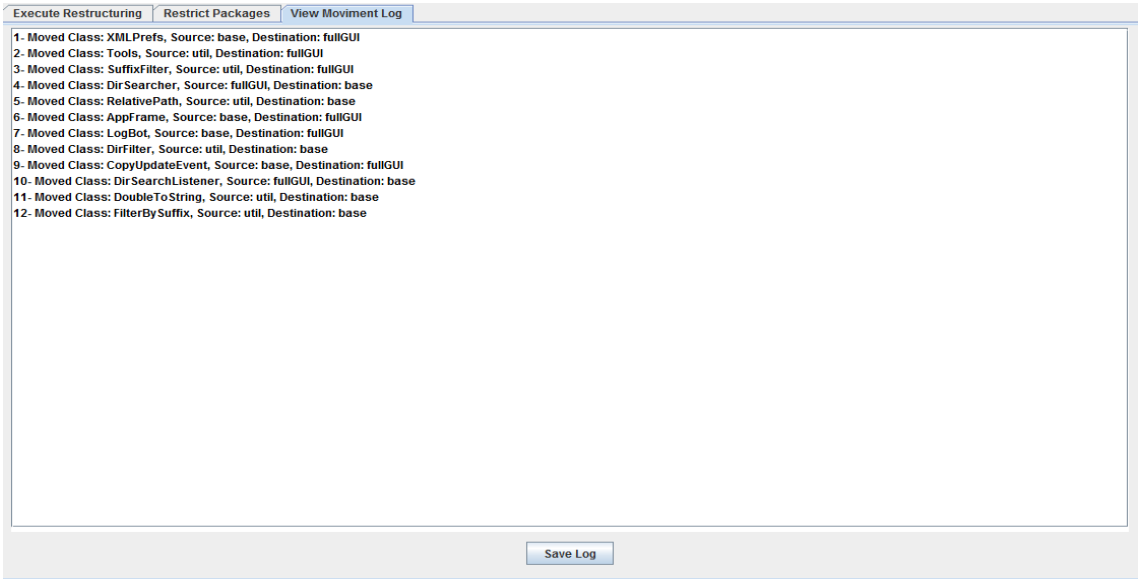
Figura 7.9 - Recorte da Tela de Restrição de Pacotes Aba *Restrict Packages*



Fonte: Do autor (2017).

Para proporcionar maior transparência e facilitar a revisão da reestruturação, em SRT, há a possibilidade de o usuário visualizar o *log* dos movimentos realizados durante a reestruturação, por meio da aba `View Moviment Log` (**8** - Figura 7.7). Esse *log* contém, para cada movimento realizado, o nome da classe movimentada e os pacotes origem e destino (Figura 7.10). No exemplo apresentado, 12 classes foram movimentadas (Figura 7.10). O usuário pode salvar esse *log* em um arquivo texto, acionando o botão `Save Log` (Figura 7.10); uma janela é aberta para selecionar um diretório para armazenar o *log*. Com o *log* de movimentação armazenado, o usuário pode atualizar a documentação do sistema de software.

Figura 7.10 - Recorte da Tela de Log de Movimentação de Classes - Aba *View Moviment Log*



Fonte: Do autor (2017).

Analisando o valor das medidas do Grupo de Avaliação e a visualização das dependências entre pacotes, apresentadas na interface de SRT (Figura 7.7), pode-se verificar que a reestruturação aprimorou o valor das medidas e reduziu a quantidade de dependência entre pacotes. Consequentemente, a modularidade e a manutenibilidade do sistema de software foram aprimoradas. Outro ponto positivo de SRT é a visualização das dependências entre pacotes, que torna a avaliação da reestruturação mais natural ao usuário, pois permite verificar, de forma intuitiva, se a modularidade do sistema de software reestruturado realmente melhorou (redução das dependências entre pacotes). Dessa forma, a visualização de dependências auxilia usuários que não possuem familiaridade com medidas de software, pois fornece uma forma gráfica de compreender o resultado da reestruturação.

7.6 Considerações finais

Neste capítulo, foi apresentada uma visão geral de SRT (***S**oftware **R**estructuring **T**ool*), um *plug-in* para a plataforma Eclipse IDE, desenvolvido para automatizar ARS (Abordagem para Reestruturação de Software). Para isso, foram detalhadas as principais tecnologias empregadas no desenvolvimento de SRT, sua organização arquitetural e o tipo abstrato de dados desenvolvido para tornar a reestruturação mais eficiente. Por fim, foi apresentado um exemplo real de utilização de SRT, para facilitar a compreensão do seu funcionamento e apresentar as demais funções presentes nesse *plug-in*.

8 AVALIAÇÃO DA ABORDAGEM

Nesse capitulo é apresentada a avaliação para verificar a eficiência da abordagem proposta para reestruturação de sistemas de software.

8.1 Considerações iniciais

Neste capítulo, são apresentadas as avaliações realizadas para verificar a eficiência da abordagem para reestruturação de sistemas de software (ARS) e comprovar se sua utilização gerou ganhos à estrutura do sistema de software. Para isso, foram conduzidas duas avaliações: i) avaliação interna, na qual foram comparados os estados antes e depois da reestruturação dos sistemas de software analisados para verificar se a abordagem aprimorou a estrutura desses sistemas; e ii) avaliação externa, na qual a estrutura sugerida por ARS foi comparada com a estrutura sugerida por abordagens apresentadas em trabalhos relacionados e do mercado.

O restante deste capítulo está organizado da seguinte maneira. Os sistemas de software utilizados para conduzir as avaliações estão caracterizados na Seção 8.2. A avaliação interna, na qual foi avaliada a eficiência de ARS, é apresentada na Seção 8.3. A avaliação externa, que comparou ARS com abordagens semelhantes existentes na literatura e no mercado é exibida na Seção 8.4. Os resultados dessas avaliações são discutidos na Seção 8.5.

8.2 Caracterização dos sistemas de software utilizados

Para realizar a avaliação de ARS, foi utilizado um conjunto de 50 sistemas de software coletados de repositórios de código aberto, como `Git`[12] e `Sourceforge`[13]. Para compor esse conjunto, o requisito principal foi o sistema ter sido utilizado por algum trabalho relacionado para avaliar seus resultados. Dessa forma, a escolha desses sistemas de software torna-se não tendenciosa à ARS e os resultados são o reflexo da sua eficácia e eficiência. Além desse requisito, para o sistema de software ser selecionado, ele deveria atender aos seguintes critérios:

a) Ser desenvolvido em Java;
b) Possuir código livre e disponível;
c) Possuir projeto para o Eclipse;

[12] http://git-scm.com/
[13] http://sourceforge.net/

d) Ter sido modificado (atualizado), no máximo, 4 anos.

Os sistemas de software selecionados são apresentados na Tabela 8.1, que contém um indexador único, nome, quantidade de pacotes, quantidade de classes de cada sistema e o(s) indexador(es) dos trabalhos que os utilizaram. Esses trabalhos têm suas referências apresentadas na Tabela 8.2 e podem ser rastreados por meio de seu indexador (#).

Tabela 8.1 - Caracterização dos Sistemas de Software Selecionados (Continua)

	Sistemas de Software	**Quantidade de Pacotes**	**Quantidade de Classes**	**Indexador do Trabalho**
S1	AndEngine	138	596	26
S2	Ant- Contrib	30	98	8; 22
S3	Antlr-IDE	129	672	8
S4	Apache-Abdera	120	678	26
S5	Apache-log4j	331	1446	15; 16
S6	Aspectj	13	70	8
S7	Atunes	67	1500	26
S8	Azures	554	3568	1; 2; 14
S9	Bluej	43	619	9
S10	Eclipse UI	1261	5955	24
S11	Findbugs	109	1160	8
S12	Freecol	61	773	8
S13	Freemind	70	491	8
S14	FrontEndForMySQL	17	98	16
S15	Fudaa	280	4216	8
S16	Ganttproject	95	614	10; 14; 23
S17	JabRef	135	857	9
S18	Jactor	32	157	9
S19	Jdk	1190	15387	23
S20	JEdit	90	573	2; 10; 22
S21	Jena	832	5090	8; 22
S22	Jfreechart	73	1017	9
S23	Jgraph	35	327	5; 8; 25
S24	JHotDraw	24	309	3; 5; 7; 9; 10; 17; 18; 19; 23
S25	Jmeter	328	1179	8
S26	Jmoney	12	54	5
S27	Jnode	1818	16848	8
S28	Joda	33	356	26
S29	Jpox	48	672	8
S30	Jtopen	18	1794	22
S31	Junit	28	162	8
S32	Jvlt	23	221	10
S33	Liferay	1659	8696	21
S34	Lucene	748	3805	8
S35	Masu	32	519	25
S36	Maven	241	965	19
S37	Openxava	144	1573	8
S38	PersonalAccess	82	457	8
S39	PHPEclipse	244	1361	8

Tabela 8.1 - Caracterização dos Sistemas de Software Selecionados (Conclusão)

	Sistemas de Software	Quantidade de Pacotes	Quantidade de Classes	Indexador do Trabalho
S40	Sapia	4	26	8
S41	Spring	999	5990	8; 21
S42	Struts	275	1922	24
S43	Swing	53	561	20
S44	Trama	8	18	16
S45	Weka	103	1568	8
S46	Xalan	81	962	22
S47	Xdoclet	106	548	21
S48	Xerces	53	757	14
S49	Xmsf	7	60	8
S50	XOM	12	254	19; 23

Fonte: Do autor (2017).

Tabela 8.2 - Trabalhos Relacionados (Continua)

#	Referência do Trabalho
1	O'KEEFFE, M.; CINNÉIDE, M. Search-based refactoring for software maintenance. **Journal of Systems and Software**, New York, v. 81, n. 4, p. 502-516, Abril de 2008.
2	ABDEEN, H.; DUCASSE, S.; SAHRAOUI, H.; ALLOUI, I. Automatic Package Coupling and Cycle Minimization. In: WORKING CONFERENCE REVERSE ENGINEERING, 16., 2009, Lille. **Proceedings**… Lille: IEEE Press, 2009. p. 103-112.
3	BAVOTA, G. L.; MARCUS, A.; OLIVETO, R. Software Re-Modularization Based on Structural and Semantic Metrics. In: WORKING CONFERENCE REVERSE ENGINEERING, 17., 2010, Boston. **Proceedings**…Washington: IEEE Computer Society, 2010. p. 195-204.
4	MOGHADAM, I. H.; CINNÉIDE, M. Code-Imp: A Tool for Automated Search-Based Refactoring. In: INTRENATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., 2011, Honolulu. **Proceedings**… Berlin: ACM, 2011. p. 41-44.
5	SHAH, S. M. A.; DIETRICH, J.; MCCARTIN, C. Making smart moves to untangle programs. In: SOFTWARE MAINTENANCE AND REENGINEERING, 16., 2012, Szeged. **Proceedings**… Washington: IEEE Computer Society, 2012. p. 359-364.
6	BAVOTA, G. L.; MARCUS, A.; OLIVETO, R. Using Structural and Semantic Measures to Improve Software Modularization. **Empirical Software Engineering**, Detroit, v. 18, n. 5, p. 901-932, Outubro de 2013.
7	ABDEEN, H.; SAHRAOUI, H.; SHATA, O.; ANQUETIL, N.; DUCASSE, S. Towards Automatically Improving Package Structure While Respecting Original Design Decisions. In: WORKING CONFERENCE REVERSE ENGINEERING, 20., 2013, Koblenz-Landau. **Proceedings**… Koblenz-Landau: IEEE Press, 2013. p. 212-221.
8	ZANETTI, M. S.; TESSONE, C. J.; SCHOLTES, I.; SCHWEITZER, F. Automated Software Remodularization Based on Move Refactoring. In: INTERNATIONAL CONFERENCE ON MODULARITY, 13., 2014, Lugano. **Proceedings**… New York: ACM, 2014. p. 73-84.
9	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, 13., 2014, Blumenau. **Anais**… Blumenau: [s. n.], 2014. p. 194-208.
10	BAVOTA, G. L.; GETHERS, M.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Improving Software Modularization Via Automated Analysis of Latent Topics and Dependencies. **ACM Transactions on Software Engineering and Methodology**, New York, v. 23, n. 2, p. 1-33, Fevereiro de 2014.

Tabela 8.2 - Trabalhos Relacionados (Conclusão)

#	Referência do Trabalho
11	HAUTUS, E. Improving Java software through package structure analysis. In: INTERNATIONAL CONFERENCE SOFTWARE ENGINEERING AND APPLICATIONS, 6., 2002, Cambridge. **Anais**… Cambridge: Acta Press, 2002. p. 1-5.
12	MITCHELL, B. S.; SPIROS M. On the automatic modularization of software systems using the bunch tool. **IEEE Transactions On Software Engineering**, Piscataway, v. 32, n. 3, p. 193-208. Março de 2006.
13	SANTOS, G.; ANQUETIL, N.; ETIEN, A.; DUCASSE, S.; VALENTE, M. T. OrionPlanning: Improving modularization and checking consistency on software architecture. In: **Working Conference of Software Visualization**, 6., 2015, Bremen. **Proceedings**…Bremen: IEEE, 2015. p. 190-194.
14	OUNI, A.; KESSENTINI, M.; SAHRAOUI, H.; BOUKADOUM, M. Maintainability defects detection and correction: a multi-objective approach. **Automated Software Engineering**, Hingham, v. 20, n. 1, p. 47-79. Março de 2013.
15	PAN, W.; BO, J; BING, L. Refactoring software packages via community detection in complex software networks. **Journal of Automation and Computing**, New York, v. 10, n. 2, p. 157-166. Abril de 2013.
16	ALKHALID, A.; ALSHAYEB, M.; MAHMOUD, S.A. Software refactoring at the package level using clustering techniques. **IET software**, Lucknow, v. 3, n. 5, p. 276-284. Junho de 2011.
17	SENG, O.; STAMMEL, J.; BURKHART, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In: CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 8., 2006, Seattle. **Proceedings**… New York: ACM, 2006. p. 1909-1916.
18	SENG, O.; BAUER, M.; BIEHL, M.; PACHE, G. Search-based improvement of subsystem decompositions. In: CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 7., 2005, Washington. **Proceedings**…New York: ACM, 2005, p. 1045-1051.
19	HARMAN, M.; LAURENCE T. Pareto optimal search based refactoring at the design level. In: CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 9, 2007, London. **Proceedings**…New York: ACM, 2007. p. 1106-1113.
20	MITCHELL, B. S.; SPIROS, M. Using Heuristic Search Techniques To Extract Design Abstractions From Source Code. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, 2., 2002, San Francisco. **Proceedings**…San Francisco: [s. n.], 2002. p. 1375-1382.
21	RATZINGER, J.; THOMAS S.; HARALD C. G. On the relation of refactorings and software defect prediction. In: INTERNATIONAL WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 8., 2008, Leipzig. **Proceedings**…New York: ACM, 2008. p. 35-38.
22	MURGIA, A.; TONELLI, R.; COUNSELL, S.; CONCAS, G.; MARCHESI, M. An empirical study of refactoring in the context of FanIn and FanOut coupling. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 18,. 2011, Limerick. **Proceedings**… Washington: IEEE Computer Society, 2011. p. 372-376.
23	MOGHADAM, I. H.; MEL, O. C. Automated refactoring using design differencing. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 16., 2012, Szeged. **Proceedings**… Washington: IEEE Computer Society, 2012. pp. 43-52.
24	SHATNAWI, R.; LI, W. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. In: **International Journal of Software Engineering and Its Applications**. v. 5, n. 4, p. 127-149. Janeiro de 2011.
25	KIMURA, S.; HIGO, Y.; IGAKI, H.; KUSUMOTO, S. Move code refactoring with dynamic analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 28., 2012, Trento. **Proceedings**… Washington: IEEE Computer Society, 2012. p. 575-578.
26	CHAPARRO, O.; BAVOTA, G.; MARCUS, A.; DI PENTA, M. On the Impact of Refactoring Operations on Code Quality Metrics. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, 14., 2012, Victoria. **Proceedings**… Washington: IEEE Computer Society. p. 456-460.

Fonte: Do autor (2017).

8.3 Avaliação interna

A avaliação interna consistiu em verificar se a estrutura sugerida por ARS aprimorou a estrutura original dos sistemas de software. Essa avaliação é chamada interna, pois compara os estados antes e depois desses sistemas. Portanto, essa avaliação não envolveu dados de outras fontes para comparação (*e.g.* outra abordagem ou outra ferramenta de reestruturação). O procedimento adotado para realizar essa avaliação consistiu dos seguintes passos. Inicialmente, os sistemas de software escolhidos e coletados dos repositórios de código aberto (Seção 8.2) foram importados para o *workspace* do Eclipse. Posteriormente, por meio de SRT, esses sistemas foram reestruturados, coletando dados antes e depois da reestruturação (*e.g.* valores das medidas de software utilizadas).

Como pode ser observado na Tabela 8.3, para cada estado, foi obtido o valor das medidas de software utilizadas (variáveis). Por existirem mais de duas variáveis e existir correlação entre elas, é recomendado aplicar análises estatísticas multivariadas (FERREIRA, 2008). Esse tipo de análise refere-se aos métodos estatísticos que analisam simultaneamente múltiplas variáveis aleatórias e inter-relacionadas. Análises estatísticas univariadas poderiam ser utilizadas nesse contexto em substituição as análises multivaridas, sendo necessário aplicá-las individualmente a cada variável. Entretanto, quando aplicadas dessa maneira, as análises univariadas não expressam a correlação existente entre as variáveis. Dessa forma, as análises univariadas podem levar a erros na análise, pois o resultado do tratamento avaliando as variáveis de forma independente pode não ser significativo, mas ser significativo quando as variáveis são analisadas conjuntamente (HAIR et al., 2009).

Por isso, nessa análise, utilizando os dados apresentados na Tabela 8.3, foram realizados testes estatísticos multivariados para verificar se ARS melhora a estrutura dos sistemas de software utilizados na avaliação. Para aplicar esses testes, foi utilizado o software `R`[14] e sua interface gráfica `RStudio`[15], utilizando um pacote específico para cada teste aplicado, os quais são descritos à medida que forem utilizados.

Para realizar alguns testes dessa análise estatística (estatística descritiva e teste de normalidade), os valores das medidas utilizadas (Tabela 8.3) foram processados algebricamente, subtraindo o valor de cada medida no estado sugerido do seu valor no estado original, obtendo a melhoria proporcionada pela reestruturação para cada medida de software.

[14] https://www.r-project.org/
[15] https://www.rstudio.com/

Por exemplo, para o software S1 em relação à medida Ca, há melhoria de 163 (258 - 95). Esse processamento foi executado para eliminar a existência de um estado original e de um estado sugerido para cada sistema (antes e depois do tratamento) e possibilitar que a análise estatística pudesse ser executada, pois esses testes trabalham sobre a diferença entre os estados e não sobre os estados antes e depois da estrutura de forma independente. O resultado desse processamento é apresentado na Tabela 8.3 na coluna intitulada melhoria, que exibe a diferença entre os estados original e o sugerido de cada medida de software utilizada para cada software.

Com base nos dados da Tabela 8.3, foi realizada uma estatística descritiva para descrever e sumarizar os dados (PETERNELLI, 2016) utilizando o pacote fBasics[16]. Os resultados podem ser visualizados na Tabela 8.4, a qual apresenta um conjunto de medidas estatísticas para cada variável analisada (medida de software utilizada).

As medidas Mínimo e Máximo apresentam o menor e o maior valores dos elementos da amostra analisada, para cada variável. Observando esses valores, percebe-se que existe grande amplitude entre o menor e o maior valores. Isso pode ser ocasionado pela variabilidade de tamanho dos sistemas de software analisados, influenciando no valor final das medidas de software e, consequentemente, na discrepância entre os valores analisados.

As medidas de dispersão 1º e 3º quartil indicam o valor limite que contém uma porcentagem de elementos abaixo do valor apresentado e outra porcentagem acima. No caso do 1º quartil, 25% e 75% dos elementos da amostra estão abaixo e acima do valor limite especificado por essa medida, respectivamente. Para o 3º quartil, 25% e 75% dos elementos da amostra estão acima e abaixo do valor limite especificado por essa medida, respectivamente. Por exemplo, para a medida Ca, 25% dos sistemas analisados tiveram melhoria nessa medida menor que 304 (1º quartil) e 25% dos sistemas tiveram melhoria nessa medida maior que 5.124,8 (3º quartil). Nota-se que, em todas as medidas analisadas, o valor do 3º quartil está distante do valor máximo, ou seja, esses valores estão bem dispersos indicando alta variabilidade entre o valor desse quartil e o valor máximo.

A medida Média é utilizada para mostrar a tendência central, indicando o centro em que se concentram os elementos da amostra analisada. Outra medida de tendência central é a Mediana que indica exatamente o valor central da amostra quando os dados estão ordenados, deixando 50% dos elementos acima e 50% abaixo do seu valor. A medida Mediana é mais "robusta" que a medida Média, pois não é influenciada por valores extremos. Com base nos dados da Tabela 8.4, percebe-se que as medidas Média e Mediana são diferentes para as

[16] https://cran.r-project.org/web/packages/fBasics/index.html

variáveis analisadas, indicando que essas variáveis possuem distribuição assimétrica (BARBETTA, 2010), logo elas não possuem distribuição normal.

Tabela 8.3 - Valor das Medidas dos Estados Original e Sugerido dos Sistemas de Software e Melhoria Obtida nas Medidas de Software (Continua)

#	Estado	Ca	Melhoria	Ce	Melhoria	$LCOM4_p$	Melhoria	CBO_p	Melhoria	MPC_p	Melhoria	RFC_p	Melhoria	TCC_p	Melhoria
S1	Original	258	163	243	162	580	64	243	162	440	315	609	162	11	24
	Sugerido	95	(63%)	81	(67%)	516	(11%)	81	(67%)	125	(72%)	447	(27%)	35	(220%)
S2	Original	617	458	490	419	56	10	490	419	673	579	1.058	41	51	89
	Sugerido	159	(74%)	71	(86%)	46	(18%)	71	(86%)	94	(86%)	643	(39%)	140	(176%)
S3	Original	11.140	7.250	9.377	7.399	404	233	9.377	7.399	12.727	10.055	14.026	7.385	441	1.294
	Sugerido	3.890	(65%)	1.978	(79%)	171	(58%)	1.978	(79%)	2.672	(79%)	6.641	(53%)	1.735	(293%)
S4	Original	5.520	3.849	5.069	3.906	454	238	5.069	3.906	10.533	8.366	12.073	3.892	447	201
	Sugerido	1.671	(70%)	1.163	(77%)	216	(52%)	1.163	(77%)	2.167	(79%)	8.181	(32%)	648	(45%)
S5	Original	12.154	3.721	7.595	3.621	1.114	379	7.594	3.620	10.487	5.100	13.065	3.493	815	549
	Sugerido	8.433	(31%)	3.974	(48%)	735	(34%)	3.974	(48%)	5.387	(49%)	9.572	(27%)	1.364	(67%)
S6	Original	22	9	20	8	45	3	20	8	22	10	297	8	19	0
	Sugerido	13	(41%)	12	(40%)	42	(7%)	12	(40%)	12	(45%)	289	(3%)	19	(0%)
S7	Original	494	364	491	363	1.453	236	491	363	521	402	914	344	31	54
	Sugerido	130	(74%)	128	(74%)	1.217	(16%)	128	(74%)	119	(77%)	570	(38%)	85	(172%)
S8	Original	4.396	2.064	3.862	2.025	244	137	3.862	2.025	6.909	3.860	10.022	1.995	381	280
	Sugerido	2.332	(47%)	1.837	(52%)	107	(56%)	1.837	(52%)	3.049	(56%)	8.027	(20%)	661	(74%)
S9	Original	8.079	5.234	6.403	4.965	5.907	1.180	6.403	4.965	7.311	5.759	6.696	4.949	35	538
	Sugerido	2.845	(65%)	1.438	(78%)	4.727	(20%)	1.438	(78%)	1.552	(79%)	1.747	(74%)	573	(1.535%)
S10	Original	7.624	5.590	6.629	5.387	799	241	6.629	5.387	11.134	9.207	12.766	5.373	347	934
	Sugerido	2.034	(73%)	1.242	(81%)	558	(30%)	1.242	(81%)	1.927	(83%)	7.393	(42%)	1.281	(269%)
S11	Original	1.050	395	918	382	560	69	918	382	1.672	834	3.158	382	269	115
	Sugerido	655	(38%)	536	(42%)	491	(12%)	536	(42%)	838	(50%)	2.776	(12%)	384	(43%)
S12	Original	4.918	2.458	3.930	2.604	295	199	3.926	2.600	7.579	5.331	7.761	2.565	134	139
	Sugerido	2.460	(50%)	1.326	(66%)	96	(67%)	1.326	(66%)	2.248	(70%)	5.196	(33%)	273	(104%)
S13	Original	161	84	148	88	67	40	148	88	453	325	1.074	88	29	14
	Sugerido	77	(52%)	60	(59%)	27	(60%)	60	(59%)	128	(72%)	986	(8%)	43	(47%)
S14	Original	52.599	34.217	46.571	34.308	3.543	2.208	46.566	34.303	56.710	40.633	56.253	34.056	741	7.466
	Sugerido	18.382	(65%)	12.263	(74%)	1.335	(62%)	12.263	(74%)	16.077	(72%)	22.197	(61%)	8.207	(1.008%)

Tabela 8.3 - Valor das Medidas dos Estados Original e Sugerido dos Sistemas de Software e Melhoria Obtida nas Medidas de Software (Continua)

#	Estado	Ca	Melhoria	Ce	Melhoria	$LCOM4_p$	Melhoria	CBO_p	Melhoria	MPC_p	Melhoria	RFC_p	Melhoria	TCC_p	Melhoria
S15	Original	10.635	6.682	7.234	6.602	311	176	7.234	6.602	9.950	9.020	11.415	6.591	265	738
	Sugerido	3.953	(63%)	632	(91%)	135	(57%)	632	(91%)	930	(91%)	4.824	(58%)	1.003	(279%)
S16	Original	6.282	4.228	5.059	4.084	528	207	5.059	4.084	6.959	5.632	8.939	3.887	924	704
	Sugerido	2.054	(67%)	975	(81%)	321	(39%)	975	(81%)	1.327	(81%)	5.052	(43%)	1.628	(76%)
S17	Original	1.184	961	1.111	934	87	51	1.111	934	1.202	1.022	1.402	935	106	494
	Sugerido	223	(81%)	177	(84%)	36	(59%)	177	(84%)	180	(85%)	467	(67%)	599	(468%)
S18	Original	8.350	6.005	7.592	6.163	15.224	1.968	7.506	6.077	11.053	8.966	9.903	5.893	160	450
	Sugerido	2.345	(72%)	1.429	(81%)	13.256	(13%)	1.429	(81%)	2.087	(81%)	4.010	(60%)	610	(281%)
S19	Original	156	77	116	80	535	49	76	40	94	40	575	40	23	4
	Sugerido	79	(49%)	36	(69%)	486	(9%)	36	(53%)	54	(43%)	535	(7%)	27	(19%)
S20	Original	36.591	16.567	28.020	16.673	4.513	1.455	28.006	16.659	39.193	24.134	35.605	16.566	640	1.863
	Sugerido	20.024	(45%)	11.347	(60%)	3.058	(32%)	11.347	(59%)	15.059	(62%)	19.039	(47%)	2.503	(291%)
S21	Original	11.521	5.268	10.462	5.588	618	141	10.462	5.588	18.173	10.990	19.541	5.585	670	843
	Sugerido	6.253	(46%)	4.874	(53%)	477	(23%)	4.874	(53%)	7.183	(60%)	13.956	(29%)	1.513	(126%)
S22	Original	2.233	1.529	2.035	1.529	91	32	2.035	1.529	3.385	2.670	5.251	1.523	230	108
	Sugerido	704	(68%)	506	(75%)	59	(35%)	506	(75%)	715	(79%)	3.728	(29%)	338	(47%)
S23	Original	2.255	1.040	1.395	986	215	61	1.395	986	2.901	2.265	2.709	986	334	119
	Sugerido	1.215	(46%)	409	(71%)	154	(28%)	409	(71%)	636	(78%)	1.723	(36%)	452	(36%)
S24	Original	13.884	4.362	11.157	4.392	765	263	11.156	4.391	17.582	6.949	19.708	4.398	451	366
	Sugerido	9.522	(31%)	6.765	(39%)	502	(34%)	6.765	(39%)	10.633	(40%)	15.310	(22%)	817	(81%)
S25	Original	13.884	4.362	11.157	4.392	765	263	11.156	4.391	17.582	6.949	19.708	4.398	451	366
	Sugerido	9.522	(31%)	6.765	(39%)	502	(34%)	6.765	(39%)	10.633	(40%)	15.310	(22%)	817	(81%)
S26	Original	29	28	26	25	43	9	26	25	67	66	155	2	9	10
	Sugerido	1	(97%)	1	(96%)	34	(21%)	1	(96%)	1	(99%)	153	(1%)	19	(105%)
S27	Original	7.902	4.797	6.133	4.608	16.683	1.374	6.133	4.608	7.631	5.986	8.624	4.431	148	1.209
	Sugerido	3.105	(61%)	1.525	(75%)	15.309	(8%)	1.525	(75%)	1.645	(78%)	4.193	(51%)	1.357	(817%)
S28	Original	3.652	2.522	3.336	2.558	134	81	3.336	2.558	8.273	6.787	7.362	2.559	164	370
	Sugerido	1.130	(69%)	778	(77%)	53	(60%)	778	(77%)	1.486	(82%)	4.803	(35%)	534	(225%)

Tabela 8.3 - Valor das Medidas dos Estados Original e Sugerido dos Sistemas de Software e Melhoria Obtida nas Medidas de Software (Continua)

#	Estado	Ca	Melhoria	Ce	Melhoria	$LCOM4_p$	Melhoria	CBO_p	Melhoria	MPC_p	Melhoria	RFC_p	Melhoria	TCC_p	Melhoria
S29	Original	11.950	7.199	9.508	7.149	270	164	9.508	7.149	29.040	22.816	17.045	7.156	452	617
	Sugerido	4.751	(60%)	2.359	(75%)	106	(61%)	2.359	(75%)	6.224	(79%)	9.889	(42%)	1.069	(137%)
S30	Original	2.000	1.891	1.900	1.892	1.034	379	1.883	1.875	2.621	2.602	7.558	1.874	433	75
	Sugerido	109	(95%)	8	(100%)	655	(37%)	8	(100%)	19	(99%)	5.684	(25%)	509	(17%)
S31	Original	402	186	356	187	98	31	356	187	533	303	1.264	186	81	52
	Sugerido	216	(46%)	169	(53%)	67	(32%)	169	(53%)	230	(57%)	1.078	(15%)	133	(65%)
S32	Original	1.519	262	1.063	202	110	45	1.063	202	1.749	467	2.437	199	175	45
	Sugerido	1.257	(17%)	861	(19%)	65	(41%)	861	(19%)	1.282	(27%)	2.238	(8%)	220	(26%)
S33	Original	606	322	271	242	8.693	119	251	222	307	273	303	222	2	140
	Sugerido	284	(53%)	29	(89%)	8.574	(1%)	29	(88%)	34	(89%)	81	(73%)	143	(5.828%)
S34	Original	36.046	17.109	16.647	15.098	3.411	500	16.567	15.018	19.121	17.190	20.291	14.997	1.719	10.556
	Sugerido	18.937	(47%)	1.549	(91%)	2.911	(15%)	1.549	(91%)	1.931	(90%)	5.294	(74%)	12.274	(614%)
S35	Original	290	216	288	216	472	56	287	215	482	368	730	215	40	28
	Sugerido	74	(74%)	72	(75%)	416	(12%)	72	(75%)	114	(76%)	515	(29%)	68	(69%)
S36	Original	9.390	4.482	4.825	3.022	721	132	4.825	3.022	9.198	6.359	8.277	3.022	288	605
	Sugerido	4.908	(48%)	1.803	(63%)	589	(18%)	1.803	(63%)	2.839	(69%)	5.255	(37%)	893	(210%)
S37	Original	454	298	313	281	1.457	44	311	279	471	428	1.207	266	35	43
	Sugerido	156	(66%)	32	(90%)	1.413	(3%)	32	(90%)	43	(91%)	941	(22%)	78	(123%)
S38	Original	4.694	4.139	4.550	4.150	273	149	4.550	4.150	7.226	6.647	8.910	4.117	87	446
	Sugerido	555	(88%)	400	(91%)	124	(55%)	400	(91%)	579	(92%)	4.793	(46%)	533	(511%)
S39	Original	32.015	25.796	29.275	25.632	884	558	29.179	25.536	37.767	32.907	36.936	25.379	641	2.886
	Sugerido	6.219	(81%)	3.643	(88%)	326	(63%)	3.643	(88%)	4.860	(87%)	11.557	(69%)	3.527	(450%)
S40	Original	11	4	7	4	16	0	7	4	16	10	96	2	9	3
	Sugerido	7	(36%)	3	(57%)	16	(0%)	3	(57%)	6	(63%)	92	(4%)	11	(22%)
S41	Original	21.912	7.579	12.170	7.377	5.705	758	12.109	7.316	14.068	8.376	15.737	7.216	867	1.508
	Sugerido	14.333	(35%)	4.793	(61%)	4.947	(13%)	4.793	(60%)	5.692	(60%)	8.521	(46%)	2.375	(174%)

Tabela 8.3 - Valor das Medidas dos Estados Original e Sugerido dos Sistemas de Software e Melhoria Obtida nas Medidas de Software (Conclusão)

#	Estado	Ca	Melhoria	Ce	Melhoria	$LCOM4_p$	Melhoria	CBO_p	Melhoria	MPC_p	Melhoria	RFC_p	Melhoria	TCC_p	Melhoria
S42	Original	24.540	11.791	16.679	11.546	1.318	457	16.679	11.546	23.679	15.640	25.974	11.349	1.559	1.282
	Sugerido	12.749	(48%)	5.133	(69%)	861	(35%)	5.133	(69%)	8.039	(66%)	14.625	(44%)	2.841	(82%)
S43	Original	560	396	463	397	501	108	463	397	639	558	1.602	382	31	18
	Sugerido	164	(71%)	66	(86%)	393	(22%)	66	(86%)	81	(87%)	1.220	(24%)	49	(55%)
S44	Original	18	9	18	9	13	4	18	9	104	64	233	9	4	2
	Sugerido	9	(50%)	9	(50%)	9	(31%)	9	(50%)	40	(62%)	224	(4%)	6	(39%)
S45	Original	2.311	1.969	2.231	2.042	1.539	167	2.127	1.938	3.070	2.806	2.659	1.938	15	16
	Sugerido	342	(85%)	189	(92%)	1.372	(11%)	189	(91%)	264	(91%)	721	(73%)	31	(106%)
S46	Original	7.488	3.402	5.890	3.338	518	183	5.872	3.320	13.682	7.451	12.996	3.326	1.612	452
	Sugerido	4.086	(45%)	2.552	(57%)	335	(35%)	2.552	(57%)	6.231	(54%)	9.670	(26%)	2.064	(28%)
S47	Original	1.245	709	1.242	709	449	111	1.242	709	2.222	1.327	4.651	705	63	118
	Sugerido	536	(57%)	533	(57%)	338	(25%)	533	(57%)	895	(60%)	3.946	(15%)	181	(185%)
S48	Original	6.029	3.513	5.240	3.544	371	205	5.240	3.544	15.286	11.202	13.992	3.518	533	421
	Sugerido	2.516	(58%)	1.696	(68%)	166	(55%)	1.696	(68%)	4.084	(73%)	10.474	(25%)	954	(79%)
S49	Original	180	170	180	170	23	10	180	170	304	286	699	170	25	9
	Sugerido	10	(94%)	10	(94%)	13	(43%)	10	(94%)	18	(94%)	529	(24%)	34	(37%)
S50	Original	216	63	159	102	209	18	159	102	386	292	2.354	102	77	25
	Sugerido	153	(29%)	57	(64%)	191	(9%)	57	(64%)	94	(76%)	2.252	(4%)	102	(32%)

Fonte: Do autor (2017).

Tabela 8.4 - Estatística Descritiva

Medidas Estatísticas	Variáveis Analisadas						
	Ca	Ce	$LCOM4_p$	CBO_p	MPC_p	RFC_p	TCC_p
Mínimo	4	4	2	4	10	2	0,1
Máximo	34.217	34.308	2.208	34.303	40.633	34.056	10.555,58
1º Quartil	304,0	251,8	49,5	236,3	408,5	233,0	43,3
3º Quartil	5.124,8	4.875,8	257,5	4.875,8	8.373,5	4.819,5	614,0
Média	4.315,8	4.231,2	311,3	4.220,2	6.412,5	4.185,1	773,7
Mediana	2.261,0	2.300,0	145,0	2.291,5	4.480,0	2.277,0	240,5
Soma	215.789,0	211.560,0	15.567,0	211.009,0	320.624,0	209.253,0	38.684,9
Erro Padrão da Média	934,3	923,7	69,1	922,4	1202,1	916,3	256,9
Variância	43.650.023	42.665.343	238.565	42.541.213	72.248.926	41.981.299	3.300.445
Desvio Padrão	6.606,8	6.531,9	488,4	6.522,4	8.499,9	6.479,3	1.816,7
Assimetria	2,7	2,8	2,5	2,8	2,2	2,8	4,1
Curtose	8,2	8,8	5,5	8,8	5,0	8,8	17,5

Fonte: Do autor (2017).

A medida Soma refere-se a soma dos elementos da amostra analisada. A medida Erro Padrão da Média estima a precisão da Média amostral em relação à média da população. Dessa forma, quanto menor esse erro, mais próximo a média da amostra analisada está da média da população ou da média de outra amostra coletada. Nos dados analisados, $LCOM4_p$ é a medida de software com menor valor para a medida Erro Padrão Médio. Portanto, sua medida Média é a mais representativa para a população; ao contrário da medida MPC_p que apresenta o maior valor para a medida Erro Padrão Médio.

A medida Variância é utilizada para mostrar a dispersão, indicando o quão distante os elementos analisados estão da média amostral (valor esperado). Quanto maior o valor da medida Variância, maior é essa distância. Dessa forma, pode-se afirmar que as variáveis analisadas possuem valores distantes da média amostral, pois as variáveis apresentam valor alto para a medida Variância. Essa alta variância pode ser explicada pela variabilidade de tamanho dos sistemas analisados (*e.g.* quantidade de classes e pacotes), pois sistemas de pequeno porte tentem a apresentar valores pequenos para as medidas de software, enquanto sistemas de grandes tentem a apresentar valores altos.

Outra medida de dispersão é o Desvio Padrão, que indica a confiabilidade do valor apresentado pela média, pois a medida Desvio Padrão apresenta o quão dispersos os elementos da amostra estão em relação ao valor da medida Média. Quanto maior o valor da medida Desvio Padrão, mais dispersos estão os dados analisados. Analisando os dados, nota-se que as variáveis analisadas apresentam valor alto para a medida Desvio Padrão. Portanto, a medida Média dessa amostra não é “confiável”, visto que o alto valor da medida Desvio Padrão indica que existem

muitos valores dispersos em relação ao valor da medida Média. Esse fato ratifica o resultado apresentado na análise da medida Variância, que indica que os valores estão distantes da média amostral. Entretanto, assim como a medida Variância, a variabilidade de tamanho dos sistemas analisados pode levar a obtenção de altos valores para a medida Desvio Padrão, pois os dados estão muito dispersos em relação à média.

A medida Assimetria, também conhecida como obliquidade, é utilizada para mostrar a simetria da distribuição de probabilidade, sendo responsável por medir o grau de afastamento que a distribuição possui em relação ao seu eixo de simetria. Esse afastamento pode não existir (distribuição simétrica) ou ocorrer para a direita ou para a esquerda da distribuição (assimetria positiva ou negativa, respectivamente). Para determinar a forma de distribuição de probabilidade de uma variável, baseia-se no valor que essa medida assume:

a) Se `Assimetria > 0`, então a distribuição tem assimetria positiva, cauda positiva mais longa (cauda à direita) e concentração de elementos à esquerda;
b) Se `Assimetria < 0`, então a distribuição tem assimetria negativa, cauda negativa mais longa (cauda à esquerda) e concentração de elementos à direita;
c) Se `Assimetria = 0`, então a distribuição é simétrica e a distribuição de elementos à esquerda e à direita é idêntica.

Analisando a assimetria das variáveis utilizadas, constata-se que elas possuem assimetria positiva, pois possuem o valor da medida Assimetria maior que zero (Tabela 8.4). Portanto, os dados nessas distribuições estão mais concentrados no lado esquerdo da distribuição, caracterizando uma distribuição de probabilidade com cauda longa à direita. Logo, os valores estão mais dispersos à direita, comprovando a grande dispersão de valores indicada pelo 3º quartil. A medida Assimetria comprova o indicado pelas medidas Média e Mediana, que as variáveis analisadas não possuem distribuição normal.

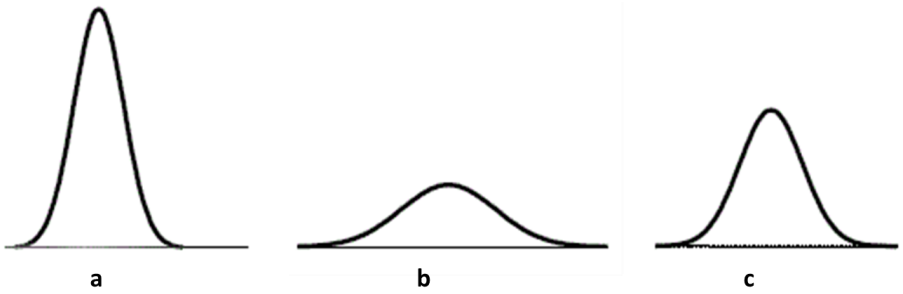
A medida Curtose indica o grau de "achatamento" da curva de distribuição de frequência dos elementos em relação a uma curva normal tomada como referência. Desse modo, a medida Curtose indica o quanto os valores da distribuição de frequência analisada estão concentrados em torno do seu centro. Quanto maior essa concentração, maior o valor dessa medida. A classificação da distribuição de frequência de acordo com a medida Curtose pode ser feita com base no coeficiente de momento dessa medida. Esse coeficiente indica que uma curva normal possui coeficiente igual a `3`; por isso, com base nesse valor tomado como referência e no valor da medida Curtose da variável analisada, pode-se caracterizar a distribuição de frequência analisada em:

a) **Leptocúrtica** (Figura 8.1-a), se `Curtose > 3`;

b) **Platicúrtica** (Figura 8.1-b), se `Curtose < 3`;

c) **Mesocúrtica** (Figura 8.1-c), se `Curtose = 3`.

Figura 8.1 - Classificação da Distribuição de Frequência de Acordo com a Curtose



Fonte: Do autor (2017).

Analisando a medida Curtose das variáveis da Tabela 8.4, constata-se que essas variáveis possuem distribuição leptocúrtica, pois seus valores para essa medida são maiores que `3`. Isso ratifica que as variáveis trabalhadas não seguem distribuição normal, pois seus valores estão concentrados em torno do seu centro (Média). A medida com maior concentração é TCC_p, enquanto MPCp é a medida com menor concentração

A estatística descritiva (Tabela 8.4) indica que as variáveis utilizadas não seguem distribuição normal, pois essas variáveis possuem distribuição de frequência leptocúrtica, assimetria positiva, Média e Mediana com valores diferentes e alto valor para as medidas Desvio Padrão e Variância. Entretanto, os resultados da estatística descritiva somente sumarizam os dados e revelam o comportamento de cada variável de forma independente. Por isso, foi utilizado o teste de normalidade multivariado *E-test* (SZÉKELY; RIZZO, 2005), com *alfa* igual a `5%` (confiança de `95%`) e disponível no pacote `energy`[17] do software `R`. Esse teste teve o objetivo de verificar se as variáveis utilizadas seguem distribuição normal multivariada, ou seja, considerando a correlação entre as variáveis. O *E-test* possui duas hipóteses:

H_0 = os dados seguem distribuição normal

H_1 = os dados não seguem distribuição normal

O resultado da execução desse teste, utilizando os dados da Tabela 8.3, indica, com `95%` de confiabilidade (nível de significância igual a `5%`), que os dados analisados não apresentam distribuição normal multivariada, porque o p-valor obtido nesse teste ($2x10^{-16}$) é menor que o nível de significância, rejeitando H_0 e aceitando H_1.

[17] https://cran.r-project.org/web/packages/energy/index.html

Para comprovar se a melhoria proporcionada aos sistemas de software reestruturados foi significativa em relação as variáveis utilizadas de forma conjunta, foi utilizado o teste estatístico multivariado *T2 de Hotelling* (FERREIRA, 2008). Esse teste é utilizado para comparar amostras emparelhadas, ou seja, amostras tomadas aos pares, antes e após a aplicação de um tratamento, sobre `n` elementos de uma população. No contexto desta pesquisa, antes e após a execução da reestruturação dos 50 sistemas de software analisados. O teste *T2 de Hotelling* possui dois pressupostos para sua execução (FERREIRA, 2008):

a) **Os dados devem possuir distribuição normal**. O teste de normalidade realizado indica que os dados trabalhados não possuem distribuição normal. Contudo, o teste *T2 de Hotelling* é robusto o suficiente para apresentar resultados confiáveis mesmo com dados que não seguem esse tipo de distribuição (COLENGHI; MINGOTI, 2008);
b) **Os dados devem provir de amostras independentes**. Os dados trabalhados nessa análise provêm de amostras independentes, pois um software não apresenta relacionamento com outro.

O teste *T2 de Hotelling* foi executado de maneira unilateral à esquerda; por isso, as hipóteses adotadas são:

$$H_0 = \mu_d = \delta_0$$

$$H_1 = \mu_d > \delta_0$$

sendo H_0 hipótese nula, em que as médias das amostras antes e depois do tratamento não diferem (são estatisticamente iguais), pois o resultado de suas diferenças é igual a variação nula, e H_1 hipótese alternativa, indicando que a diferença existente entre as amostras é maior que a variação nula. Nessas hipóteses, μ_d representa a diferença entre as médias amostrais ($\mu_d = \mu_2 - \mu_1$, em que μ_1 e μ_2 referem-se à média da variável antes e depois da aplicação do tratamento, respectivamente) e δ_0 representa a variação nula entre as amostras.

Para aplicar o teste *T2 de Hotelling*, foi utilizada a função `HotellingsT2Test` do pacote `DescTolls`[18] do software `R` que utiliza duas tabelas como entrada para calcular o valor do teste. Uma tabela possui o valor das medidas de software no estado original (antes da reestruturação) e outra tabela possui o valor das medidas de software no estado sugerido (após a reestruturação). Portanto, essa função calcula a diferença (melhoria apresentada na Tabela 8.3) automaticamente para obter o resultado do teste *T2 de Hotelling*.

[18] https://cran.r-project.org/web/packages/DescTools/index.html

O resultado do teste *T2 de Hotelling* indica que H_1 é verdadeira, rejeitando H_0 (Figura 8.2). Esse resultado é confirmado, pois o p-valor para o teste ($9,0 \times 10^{-6}$) é menor que o nível de significância trabalhado (5%). Portanto, existem evidências significativas que há efeito expressivo da reestruturação sobre as medidas de software analisadas de forma correlacionada. Dessa maneira, pode-se afirmar que os valores das medidas de software na estrutura sugerida são melhores que os presentes na estrutura original. Esse resultado foi ratificado com a função `hotelling.test` do pacote `Hotelling`[19], que também executa o teste *T2 de Hotelling* e apresentou o mesmo resultado.

Figura 8.2 - Resultado Teste *T2 de Hotelling*

```
Hotelling's two sample T2-test

data:  dadosOriginais and dadosSugeridos
T.2 = 6, df1 = 7, df2 = 90, p-value = 0.000009
alternative hypothesis: true location difference is not equal to c(0,0,0,0,0,0,0)
```

Fonte: Do autor (2017).

Visto que H_0 foi rejeitada, então deve-se saber quais variáveis analisadas (medidas de software) foram responsáveis. Como são trabalhadas mais de 3 variáveis, podem ser construídos intervalos de confiança simultâneos para saber qual(is) é(são) essa(s) variável(is). A construção desses intervalos de confiança (IC) é necessária, pois análises multivariadas realizam afirmações considerando a correlação entre as variáveis. Portanto, pode-se ter variáveis não responsáveis pela rejeição de H_0, sendo essencial conhecer esse fato para saber a real eficácia de ARS. Para realizar essa tarefa, não foi encontrada uma função pronta no software `R`; por isso, a seguinte fórmula foi transcrita para esse software para possibilitar sua realização (FERREIRA, 2008):

$$IC_{1-\alpha}(l^T \mu_d): l^T \, \overline{D} \pm \sqrt{\frac{vp}{v+1-p} F_{\alpha,p,v+1-p}} \sqrt{\frac{l^T \, S_d l}{n}}$$

sendo l^T o vetor de combinação linear, que contém `1` na posição que se deseja comparar as médias e `0` nas demais, $\overline{D}$ o vetor de médias, $S_d l$ a matriz de covariância da amostra, `v` os graus de liberdade do teste, `p` a quantidade de variáveis analisadas, `n` a quantidade de observações da amostra e $F_{\alpha,p,v+1-p}$ o valor da distribuição `F` para a amostra trabalhada.

O resultado da construção dos intervalos de confiança simultâneos é apresentado na Tabela 8.5, revelando os valores mínimo e máximo desses intervalos em relação a cada variável

[19] https://cran.r-project.org/web/packages/Hotelling/index.html

utilizada. Para compreender o resultado desses intervalos e verificar se a variável foi responsável pela rejeição de H_0, deve ser verificado se as coordenadas do ponto `[0,0,0,0,0,0]`^{T} estão contidas em todos os intervalos da variável. Em caso positivo, essa variável aceita H_0. Caso contrário, se pelo menos um dos intervalos não conter a coordenada `0`, então essa variável é responsável pela rejeição de H_0.

Tabela 8.5 - Intervalos de Confiança Simultâneos

Variáveis Analisadas	Intervalos	Variáveis Analisadas						
		Ca	**Ce**	**$LCOM4_p$**	**CBO_p**	**MPC_p**	**RFC_p**	**TCC_p**
Ca	Mínimo	358	383	3.427	386	-75	399	2.480
	Máximo	8.274	8.249	5.205	8.246	8.707	8.233	6.152
Ce	Mínimo	298	318	3.343	320	-142	333	2.434
	Máximo	8.164	8.144	5.119	8.142	8.604	8.129	6.028
$LCOM4_p$	Mínimo	-578	-577	18	-576	-658	-572	-81
	Máximo	1.200	1.199	604	1.198	1.280	1.194	703
CBO_p	Mínimo	290	309	3.333	312	-150	325	2.424
	Máximo	8.150	8.131	5.107	8.128	8.590	8.115	6.016
MPC_p	Mínimo	2.021	2.039	-5.443	2.042	1.319	2.055	4.479
	Máximo	10.803	10.785	7.381	10.782	11.505	10.769	8.345
RFC_p	Mínimo	268	287	3302	290	-172	303	2.394
	Máximo	8.102	8.083	5.068	8.080	8.542	8.067	5.976
TCC_p	Mínimo	-6.972.545	-6.972.506	-6.971.101	-6.972.505	-6.972.642	-6.972.500	-6.971.797
	Máximo	-6.968.873	-6.968.912	-6.970.317	-6.968.913	-6.968.776	-6.968.918	-6.969.621
Resultado		**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**

Fonte: Do autor (2017).

Como pode ser observado na última linha da Tabela 8.5, os intervalos de confiança simultâneos revelam que as sete variáveis analisadas são responsáveis pela rejeição de H_0. Isso indica que as sete medidas de software utilizadas (Ca, Ce, $LCOM4_p$ CBO_p, MPC_p, RFC_p e TCC_p) foram modificadas significativamente entre os estados antes e depois da reestruturação, tendo seus valores aprimorados significativamente. Outros dados foram coletados do processo de reestruturação, permitindo avaliar outros ângulos de desempenho de ARS (Tabela 8.6). O tempo médio gasto para realizar as reestruturações foi **2 horas 55 minutos e 56 segundos,** sendo **5 minutos e 37 segundos** no melhor caso (`S29`) e **13 horas e 43 minutos** no pior caso (`S24`).

Ao avaliar se o tamanho do sistema de software analisados (quantidade de classes) interfere na quantidade de classes movimentadas por ARS (Figura 8.3), o resultado é negativo, pois a porcentagem de classes movimentas varia (aumenta e reduz), à medida que o tamanho

dos sistemas de software aumenta. Sendo assim, a quantidade de classes movimentadas altera em função do quanto a estrutura do sistema de software está degradada e não em função do seu tamanho. Isso indica que a abordagem realiza movimentos de acordo com o quanto o sistema de software pode ser aprimorado e não em relação ao seu tamanho.

Tabela 8.6 - Dados do Processo de Reestruturação (Continua)

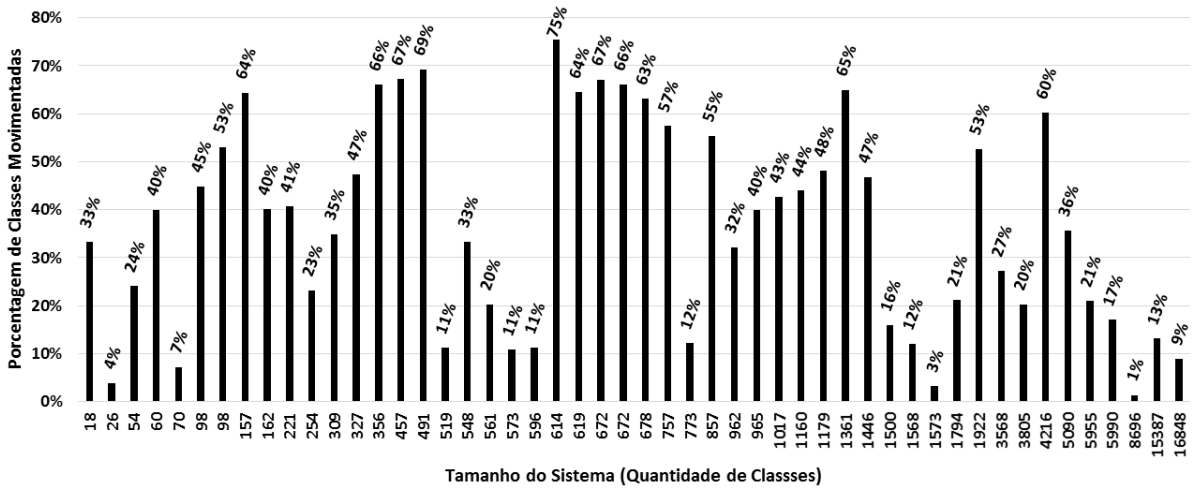
#	Tempo de Reestruturação (hh:mm:ss)	Quantidade de Classes Movimentadas	Quantidade de Movimentos	Quantidade de Dependência Entre Pacotes - Estrutura Original	Quantidade de Dependências Entre Pacotes - Estrutura Sugerida
S1	2:41:00	67	67	991	299
S2	1:52:50	44	44	1.434	268
S3	0:36:34	451	451	32.016	8.053
S4	4:52:46	428	428	23.905	4.819
S5	0:33:12	677	677	33.298	19.592
S6	1:43:03	5	5	45	27
S7	0:29:53	239	239	1.205	296
S8	3:55:51	974	974	29.820	13.683
S9	1:10:51	399	399	15.149	6.916
S10	1:20:17	1.251	1.251	20.308	6.015
S11	3:38:39	510	510	22.994	5.118
S12	1:30:57	94	94	3.457	1.775
S13	2:19:58	340	340	16.236	5.948
S14	0:56:21	52	52	836	289
S15	2:55:38	2.536	2.536	145.243	43.949
S16	0:33:17	463	463	27.174	7.188
S17	0:21:05	475	475	16.855	4.791
S18	0:20:57	101	101	2.442	412
S19	0:08:47	2.019	2.019	27.418	5.720
S20	0:34:04	62	62	432	165
S21	1:28:19	1.817	1.817	102.385	47.644
S22	1:37:24	434	434	38.088	18.118
S23	1:25:00	155	155	7.234	3.052
S24	13:43:00	108	108	6.956	1.637
S25	8:51:00	567	567	42.340	27.407
S26	8:45:00	13	13	136	2
S27	8:54:00	1.487	1.487	23.845	6.187
S28	1:28:01	235	235	23.252	4.293
S29	0:05:37	444	444	65.440	19.672
S30	6:08:00	379	379	5.787	145
S31	0:24:30	65	65	1.129	511
S32	9:23:00	90	90	4.112	3.187
S33	4:30:00	120	120	1.250	494
S34	11:02:00	773	773	71.634	31.697
S35	1:47:06	58	58	896	216
S36	9:47:00	385	385	28.224	12.867
S37	2:05:22	50	50	954	209
S38	5:35:20	307	307	16.263	1347
S39	0:15:20	884	884	92.367	14.299

Tabela 8.6 - Dados do Processo de Reestruturação (Conclusão)

#	Tempo de Reestruturação (hh:mm:ss)	Quantidade de Classes Movimentadas	Quantidade de Movimentos	Quantidade de Dependência Entre Pacotes - Estrutura Original	Quantidade de Dependências Entre Pacotes - Estrutura Sugerida
S40	0:24:35	1	1	46	15
S41	1:22:34	1.020	1.020	54.721	33.624
S42	1:17:44	1.013	1.013	69.781	34.840
S43	0:25:28	114	114	1.626	340
S44	0:43:17	6	6	184	83
S45	0:57:51	189	189	7.638	1.140
S46	0:41:31	310	310	29.449	14.040
S47	4:52:00	183	183	4.849	1.825
S48	4:08:52	435	435	33.050	10.072
S49	1:15:32	24	24	689	30
S50	1:58:51	59	59	1.362	434

Fonte: Do autor (2017).

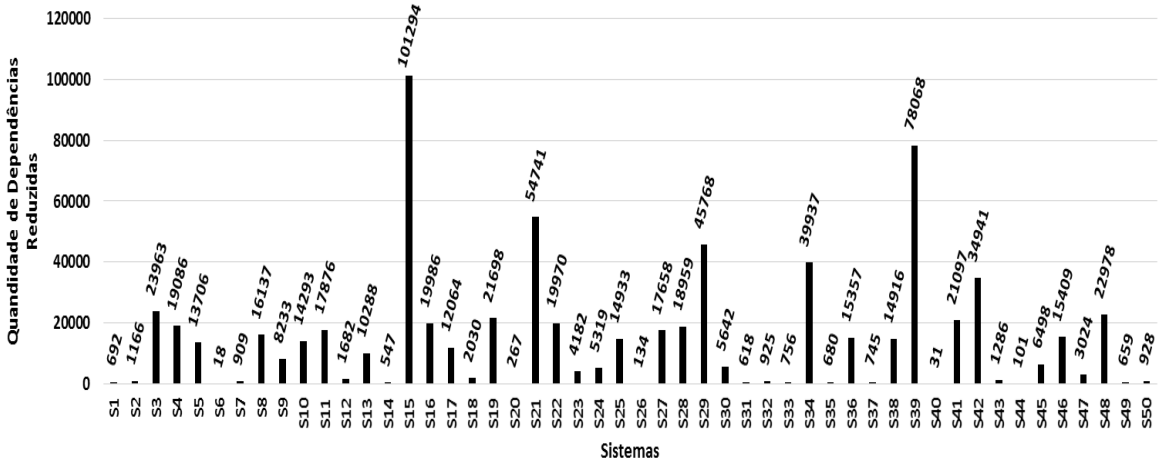
Figura 8.3 - Relação Entre a Porcentagem de Classes Movimentadas e o Tamanho dos Sistemas de Software



Fonte: Do autor (2017).

Analisando a quantidade de dependências entre pacotes existente nos sistemas de software analisados (Tabela 8.6), pode-se afirmar que em todos os casos essas dependências foram reduzidas. Isso pode ser visualizado mais claramente na Figura 8.4, na qual o resultado da diferença entre a quantidade de dependências entre pacotes existente nas estruturas original e sugerida é apresentado para cada sistema de software. Por exemplo, o software `S1` na estrutura original, há `991` dependências entre pacotes e, na estrutura sugerida, há `299` dependências entre pacotes; logo, a diferença é `692`. Portanto, esses sistemas de software tornaram-se mais modulares, pois suas dependências entre pacotes foram reduzidas e, consequentemente, suas dependências intrapacotes aumentaram.

Figura 8.4 - Quantidade de Dependências Entre as Estruturas Original e Sugerida



Fonte: Do autor (2017).

8.4 Avaliação externa

A avaliação externa consistiu em comparar ARS com as abordagens apresentadas por trabalhos relacionados e no mercado, realizando uma comparação entre tratamentos para verificar a existência de diferença entre os efeitos dos tratamentos realizados (BARBETTA, 2010). Para permitir essa comparação, uma busca por ferramentas de reestruturação de software foi realizada, encontrando cinco ferramentas:

a) **`Code-Imp`** (MOGHADAM; CINNÉIDE, 2011). Essa ferramenta é um *plug-in* para a plataforma Eclipse IDE que realiza diversos tipos de reestruturações simultâneas sobre código Java, como movimentar e alterar visibilidade de métodos. Para verificar se a reestruturação conduzida aperfeiçoou a estruturada do sistema de software, um conjunto de 28 medidas de software foi utilizado, em que a soma dos valores dessas medidas determina se houve melhoria na nova estrutura, podendo as medidas receberem pesos de acordo com os critérios do usuário. O processo de reestruturação dessa ferramenta é apoiado por duas variações da meta-heurísticas *hill climbing: first-ascent hill-climbing* e *steepest-ascent hillclimbing*. Essa ferramenta não está disponível para *download*, portanto, não foi possível utilizá-la;

b) **`R3`** (BAVOTA et al., 2014). Essa ferramenta é independente de plataforma e realiza remodularizações baseando-se na movimentação de classes entre pacotes. Para identificar as possibilidades de movimento, `R3` utiliza medidas estruturais (dependências entre classes) identificadas por meio de análise do código do sistema de software, medidas semânticas (responsabilidades implementadas por uma classe)

identificadas por meio de identificadores, comentários e outras *strings* existentes no código fonte e na técnica RTM (*Relational Topic Model*). Com base nesses dois tipos de medidas, essa ferramenta analisa o ponto de vista conceitual (medidas semânticas) e estrutural (medidas estruturais) do código, armazenando o resultado das análises em matrizes. Posteriormente, essas matrizes são utilizadas como entrada para a RTM, que analisa e sugere movimentos de classes entre pacotes, permitindo mover classes para pacotes com maior quantidade de classes similares. Essa ferramenta não está disponível para *download*, portanto, não foi possível utilizá-la;

c) **CARE**[20]**.** Essa ferramenta é um *plug-in* para a plataforma Eclipse IDE, cujo objetivo é aprimorar a modularização de sistemas de software. Entretanto, o processo utilizado por CARE para realizar a remodularização não é revelado. Essa ferramenta não está disponível para *download*, portanto, não foi possível utilizá-la;

d) **SOMOMOTO**[21]**.** Essa ferramenta é um *plug-in* para a plataforma Eclipse IDE, cujo objetivo é remodularizar sistemas de software Java. Para realizar a remodularização, SOMOMOTO baseia-se no movimento de classes entre pacotes. Para isso, a cada iteração, uma classe do sistema de software é escolhida aleatoriamente, suas dependências intra e entre pacotes são analisadas e é determinada a probabilidade dessa classe ser movimenta para o pacote com o qual a classe estiver mais conectada. O objetivo é realocá-la em melhor posição na estrutura do sistema de software. Essa ferramenta está disponível para *download*, mas apresenta falhas ao "carregar" suas classes, impossibilitando sua utilização. Os desenvolvedores dessa ferramenta foram contatados, mas estão com indisponibilidade de consertar o problema existente, inviabilizando sua utilização;

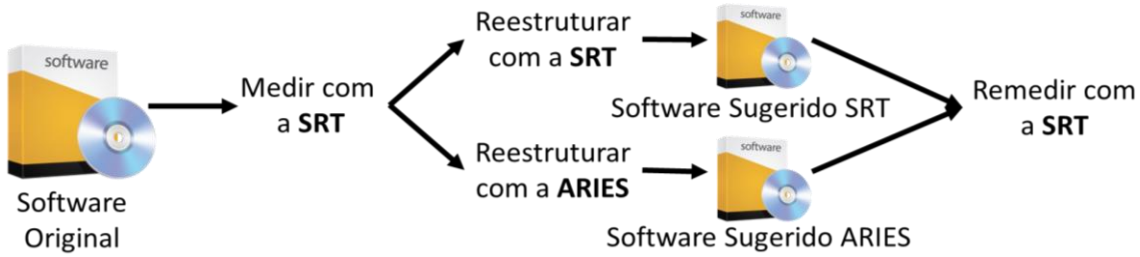
e) **ARIES**[22]. Esse *plug-in* para a plataforma Eclipse IDE realiza a refatoração de código Java, dividindo pacotes em unidades mais coesas ou dividindo classes em unidades mais responsáveis (com somente uma funcionalidade) cabendo ao usuário escolher em qual granularidade a análise deve ser executada. Para alcançar esse objetivo, foi utilizado o apoio das medidas de software *Information flow-based coupling* (ICP) e *Conceptual Coupling Between Classes* (CCBC), para identificar pacotes que possuem classes pouco relacionadas e indicar pontos em que as reestruturações devem ser aplicadas. Essa

[20] https://sites.google.com/site/careplugin/home
[21] https://sourceforge.net/projects/somomoto/
[22] http://www.sesa.unisa.it/tools/aries.jsp

ferramenta está disponível para *download* e funciona adequadamente, sendo utilizada para realizar a avaliação externa.

Dessa forma, ARIES foi utilizada para realizar a avaliação. Mesmo ARIES apresentando uma abordagem de reestruturação diferente de SRT, pode-se comparar o estado da estrutura sugerida por essas ferramentas, por meio dos valores das medidas de software utilizadas nesta investigação. Dessa maneira, é necessário que essas medidas sejam coletadas da estrutura sugerida por ARIES para realizar a comparação. Para isso, o seguinte procedimento foi empregado (Figura 8.5). O sistema de software é medido com SRT para coletar o valor das medidas de software utilizadas nesta pesquisa. Esse sistema é reestruturado utilizando SRT e ARIES, obtendo duas estruturas sugeridas (Software Sugerido SRT e Software Sugerido ARIES, respectivamente). Em seguida, essas duas sugestões são medidas utilizando SRT para coletar o valor das medidas de software. Por fim, os valores coletados das duas estruturas sugeridas são comparados e analisados.

Figura 8.5 - Condução da Avaliação Externa



Fonte: Do autor (2017).

Esse procedimento foi realizado em um subconjunto de sistemas de software utilizado na Avaliação Interna (Tabela 8.1), pois ARIES apresentou problemas durante a reestruturação, não gerando a estrutura sugerida para ser comparada com a sugerida por SRT. Os problemas detectados durante a reestruturação foram: 1) o sistema superou cinco horas de processamento, não revelando se ARIES ainda estava processando ou havia deixado de processar; 2) ARIES por mais de duas iterações consecutivas sugeriu a divisão de um mesmo pacote, não realizando a divisão e sugerindo a mesma divisão na iteração posterior; e 3) ARIES processou por mais de 24 horas sem retornar resultado da reestruturação. Dessa maneira, alguns sistemas de software foram desconsiderados nessa avaliação, pois não foi possível obter a estrutura sugerida por ARIES para a comparar com a estrutura sugerida por SRT. Na Tabela 8.7, na coluna intitulada “Analisado” é descrito se o sistema de software foi ou não analisado por ARIES. Em caso negativo, na coluna intitulada “Motivo”, é apresentada a justifica que levou ao “descarte” desse sistema de software, apresentando os valores 1, 2 ou 3, em referência aos motivos apresentados anteriormente.

Tabela 8.7 - Sistemas de Software Utilizados na Avaliação Externa

#	Software	Analisado	Motivo	#	Software	Analisado	Motivo
S1	AndEngine	Sim	-	S26	Jmoney	Sim	-
S2	Ant- Contrib	Sim	-	S27	Jnode	Não	3
S3	Antlr-IDE	Sim	-	S28	Joda	Sim	-
S4	Apache-Abdera	Não	2	S29	Jpox	Sim	-
S5	Apache-log4j	Sim	-	S30	Jtopen	Não	3
S6	Aspectj	Sim	-	S31	Junit	Sim	-
S7	Atunes	Sim	-	S32	Jvlt	Sim	-
S8	Azures	Não	1	S33	Liferay	Não	1
S9	Bluej	Sim	-	S34	Lucene	Não	3
S10	Eclipse UI	Sim	-	S35	Masu	Não	2
S11	Findbugs	Não	1	S36	Maven	Sim	-
S12	Freecol	Não	1	S37	Openxava	Não	2
S13	Freemind	Não	2	S38	PersonalAccess	Sim	-
S14	FrontEndForMySQL	Sim	-	S39	PHPEclipse	Não	2
S15	Fudaa	Não	1	S40	Sapia	Sim	-
S16	Ganttproject	Não	2	S41	Spring Framework	Não	1
S17	JabRef	Sim	-	S42	Struts	Sim	-
S18	Jactor	Sim	-	S43	Swing	Não	2
S19	Jdk	Não	1	S44	Trama	Sim	-
S20	Jedit	Sim	-	S45	Weka	Não	1
S21	Jena	Não	1	S46	Xalan	Não	2
S22	Jfreechart	Não	1	S47	Xdoclet	Sim	-
S23	Jgraph	Não	2	S48	Xerces	Não	2
S24	JhotDraw	Sim	-	S49	Xmsf	Sim	-
S25	Jmeter	Sim	-	S50	XOM	Sim	-

Fonte: Do autor (2017). Legenda: "-" indica que o conteúdo da coluna não se aplica ao sistema de software da linha

Dessa maneira, dos 50 sistemas de software utilizados para conduzir essa avaliação, 23 foram descartados. Com base nos 27 sistemas de software restantes, foi realizada a comparação entre as estruturas sugeridas por ARIES e por SRT. Na Tabela 8.8, são apresentados os estados original e sugeridos por SRT e por ARIES dos sistemas de software analisados, para cada medida de software utilizada. Com base nos valores dessas medidas foi calculada a diferença entre seus valores (valor abaixo das células intitulada Diferença na Tabela 8.8). Por exemplo, em relação à medida Ca para o sistema de software `S1`, seu valor na estrutura sugerida por SRT é `95` e seu valor na estrutura sugerida por ARIES é `258`. Logo, a diferença entre as estruturas sugeridas é `-163` (`95 - 258`). Esse procedimento foi executado pois, nessa comparação, o importante é analisar a diferença entre os estados sugeridos pelas ferramentas e não a estrutura sugerida de forma independente. As análises estatísticas posteriores utilizam essa diferença para determinar se a estrutura sugerida por uma ferramenta foi melhor ou pior que a estrutura sugerida pela outra.

Com base nas diferenças entre os valores das medidas de software, uma estatística descritiva utilizando o pacote `fBasics` do software `R` foi realizada para descrever e resumir os dados. O resultado é apresentado na Tabela 8.9, em que 27 observações foram analisadas (quantidade de sistemas de software utilizados na avaliação externa). As medidas Máximo e Mínimo auxiliam na compreensão do intervalo em que os valores de cada variável está compreendido. Nos dados analisados, a variável com maior intervalo é a medida TCC_p; portanto, essa é a variável que apresenta a maior dispersão dos seus valores em relação as variáveis analisadas. Por outro lado, a medida $LCOM4_p$ é a variável com menor intervalo e com menor dispersão dentre as variáveis analisadas. Essa alta dispersão pode ser explicada pela variabilidade de tamanho dos sistemas de software analisados, em que sistemas de grande porte podem apresentar alto valor para as medidas analisadas e sistemas de pequeno porte podem apresentar valores baixos. As medidas 1º quartil e 3º quartil revelam que a medida $LCOM4_p$ é a variável com a maior intervalo entre o 1º quartil e seu valor mínimo e a medida TCC_p é a variável com maior intervalo entre o 3º quartil e o seu valor máximo. Portanto, essas variáveis apresentam alta dispersão dos seus valores em relação à medida Média, confirmando o indicado pelas medidas Mínimo e Máximo. Com exceção da medida TCC_p, as demais medidas apresentam 3º quartil próximo ao valor máximo, o que pode indicar uma concentração dos valores analisados perto do valor máximo identificado.

A medida Média indica que as estruturas sugeridas por SRT tiveram melhor valor do que as sugeridas por ARIES, pois a média resultante da diferença do valor das variáveis analisadas dessas estruturas foi negativa. Entretanto, não se pode fazer afirmações com base na medida Média, pois essa medida não expressa a dispersão dos valores da variável. Somente com base na medida Média, não é possível afirmar que SRT foi melhor que ARIES, pois podem existir situações em que ARIES foi melhor do que SRT, mas que não são detectáveis pela análise do valor médio. Complementando a análise da medida Média com base nos valores da medida Mediana, pode-se concluir que as variáveis analisadas possuem distribuição assimétrica, pois seus valores médios são diferentes de seus valores medianos (BARBETTA, 2010). Isso é um indicativo que essas variáveis não possuem distribuição normal.

A medida Soma apresentou valor negativo em todas as variáveis analisadas, revelando que os aprimoramentos proporcionados por SRT na estrutura dos sistemas de software analisados foram melhores do que os proporcionados por ARIES, pois o resultado, a diferença entre as estruturas sugeridas (`estrutura sugeria por ARIES – estrutura sugerida por SRT`), apresentou valor negativo.

Tabela 8.8 - Estados Original e Sugerido por SRT e por ARIES e Diferença entre esses Estados dos Sistemas de Software Analisados (Continua)

#	Estrutura	Ca		Ce		$LCOM4_p$		CBO_p		MPC_p		RFC_p		TCC_p	
	Original	258	**Dif.**	243	**Dif.**	580	**Dif.**	243	**Dif.**	440	**Dif.**	609	**Dif.**	11	**Dif.**
S1	Sugerida SRT	95	-163	81	-162	516	-64	81	-162	125	-315	447	-162	35	-24
	Sugerida ARIES	258	(63%)	243	(67%)	580	(11%)	243	(67%)	440	(72%)	609	(27%)	11	(218%)
	Original	617	**Dif.**	490	**Dif.**	56	**Dif.**	490	**Dif.**	673	**Dif.**	1.058	**Dif.**	51	**Dif.**
S2	Sugerida SRT	159	-458	71	419	46	-10	71	-419	94	-579	643	-415	140	-89
	Sugerida ARIES	617	(74%)	490	(9%)	56	(18%)	490	(86%)	673	(86%)	1.058	(39%)	51	(175%)
	Original	11.140	**Dif.**	9.377	**Dif.**	404	**Dif.**	9.377	**Dif.**	12.727	**Dif.**	14.026	**Dif.**	441	**Dif.**
S3	Sugerida SRT	3.890	-7.250	1.978	-7.399	171	-233	1.978	-7.399	2.672	-10.055	6.641	-7.385	1.735	-1.294
	Sugerida ARIES	11.140	(65%)	9.377	(79%)	404	(58%)	9.377	(79%)	12.727	(79%)	14.026	(53%)	441	(293%)
	Original	12.154	**Dif.**	7.595	**Dif.**	1.114	**Dif.**	7.594	**Dif.**	10.487	**Dif.**	13.065	**Dif.**	815	**Dif.**
S5	Sugerida SRT	8.433	-2.374	3.974	-2.406	735	-430	3.974	-2.406	5.387	-3.808	9.572	-1.026	1.364	-719
	Sugerida ARIES	10.807	(22%)	6.380	(38%)	1.165	(37%)	6.380	(38%)	9.195	(41%)	10.598	(10%)	645	(111%)
	Original	22	**Dif.**	20	**Dif.**	45	**Dif.**	20	**Dif.**	22	**Dif.**	297	**Dif.**	19	**Dif.**
S6	Sugerida SRT	13	-9	12	-8	42	-3	12	-8	12	-10	289	-8	19	0
	Sugerida ARIES	22	(41%)	20	(40%)	45	(7%)	20	(40%)	22	(45%)	297	(3%)	19	(0%)
	Original	494	**Dif.**	491	**Dif.**	1.453	**Dif.**	491	**Dif.**	521	**Dif.**	914	**Dif.**	31	**Dif.**
S7	Sugerida SRT	130	-9.552	128	-9.400	1.217	-297	128	-9.398	119	-12.898	570	-17.970	85	-465
	Sugerida ARIES	9.682	(99%)	9.528	(99%)	920	(32%)	9.526	(99%)	13.017	(99%)	18.540	(97%)	550	(85%)
	Original	4.396	**Dif.**	3.862	**Dif.**	244	**Dif.**	3.862	**Dif.**	6.909	**Dif.**	10.022	**Dif.**	381	**Dif.**
S9	Sugerida SRT	2.332	-2.064	1.837	-2.025	107	-137	1.837	-2.025	3.049	-3.860	8.027	-1.995	661	-280
	Sugerida ARIES	4.396	(47%)	3.862	(52%)	244	(56%)	3.862	(52%)	6.909	(56%)	10.022	(20%)	381	(73%)
	Original	8.079	**Dif.**	6.403	**Dif.**	5.907	**Dif.**	6.403	**Dif.**	7.311	**Dif.**	6.696	**Dif.**	35	**Dif.**
S10	Sugerida SRT	2.845	-5.234	1.438	-4.965	4.727	-1.180	1.438	-4.965	1.552	-5.759	1.747	-4.949	573	-538
	Sugerida ARIES	8.079	(65%)	6.403	(78%)	5.907	(20%)	6.403	(78%)	7.311	(79%)	6.696	(74%)	35	(1537%)
	Original	161	**Dif.**	148	**Dif.**	67	**Dif.**	148	**Dif.**	453	**Dif.**	1.074	**Dif.**	29	**Dif.**
S14	Sugerida SRT	77	-84	60	-88	27	-40	60	-88	128	-325	986	-88	43	-895
	Sugerida ARIES	161	(52%)	148	(59%)	67	(60%)	148	(59%)	453	(72%)	1.074	(8%)	29	(48%)
	Original	6.282	**Dif.**	5.059	**Dif.**	528	**Dif.**	5.059	**Dif.**	6.959	**Dif.**	8.939	**Dif.**	924	**Dif.**
S17	Sugerida SRT	2.054	-4.163	975	-4.013	321	-204	975	-4.013	1.327	-5.554	5.052	-3.757	1.628	-738
	Sugerida ARIES	6.217	(67%)	4.988	(80%)	525	(39%)	4.988	(80%)	6.881	(81%)	8.809	(43%)	890	(83%)

Tabela 8.8 - Estados Original e Sugerido por SRT e por ARIES e Diferença entre esses Estados dos Sistemas de Software Analisados (Continua)

#	Estrutura	Ca		Ce		$LCOM4_p$		CBO_p		MPC_p		RFC_p		TCC_p	
	Original	1.184	**Dif.**	1.111	**Dif.**	87	**Dif.**	1.111	**Dif.**	1.202	**Dif.**	1.402	**Dif.**	106	**Dif.**
S18	Sugerida SRT	223	-901	177	-934	36	-51	177	-934	180	-1.022	467	-935	599	-493
	Sugerida ARIES	1.184	(81%)	1.111	(84%)	87	(59%)	1.111	(84%)	1.202	(85%)	1.402	(67%)	106	(465%)
	Original	156	**Dif.**	116	**Dif.**	535	**Dif.**	76	**Dif.**	94	**Dif.**	575	**Dif.**	23	**Dif.**
S20	Sugerida SRT	79	-77	36	-80	486	-49	36	-40	54	-40	535	-40	27	-4
	Sugerida ARIES	156	(49%)	116	(69%)	535	(9%)	76	(53%)	94	(43%)	575	(7%)	23	(17%)
	Original	2.233	**Dif.**	2.035	**Dif.**	91	**Dif.**	2.035	**Dif.**	3.385	**Dif.**	5.251	**Dif.**	230	**Dif.**
S24	Sugerida SRT	704	-1.529	506	-1.529	59	-32	506	-1.529	715	-2.670	3.728	-1.523	338	-108
	Sugerida ARIES	2.233	(68%)	2.035	(75%)	91	(35%)	2.035	(75%)	3.385	(79%)	5.251	(29%)	230	(47%)
	Original	13.884	**Dif.**	11.157	**Dif.**	765	**Dif.**	11.156	**Dif.**	17.582	**Dif.**	19.708	**Dif.**	450	**Dif.**
S25	Sugerida SRT	9.522	-2.655	6.765	-2.971	502	-245	6.765	-2.971	10.633	-5.231	15.310	-2.449	817	-390
	Sugerida ARIES	12.177	(22%)	9.736	(31%)	747	(33%)	9.736	(31%)	15.864	(33%)	17.759	(14%)	427	(91%)
	Original	29	**Dif.**	26	**Dif.**	43	**Dif.**	26	**Dif.**	67	**Dif.**	145	**Dif.**	9	**Dif.**
S26	Sugerida SRT	1	-28	1	-25	34	-9	1	-25	1	-66	143	-2	19	-10
	Sugerida ARIES	29	(97%)	26	(96%)	43	(21%)	26	(96%)	67	(99%)	145	(1%)	9	(111%)
	Original	3.652	**Dif.**	3.336	**Dif.**	134	**Dif.**	3.336	**Dif.**	8.273	**Dif.**	7.362	**Dif.**	164	**Dif.**
S28	Sugerida SRT	1.130	-2.522	778	-2.558	53	-81	778	-2.558	1.486	-6.787	4.803	-2.559	534	-370
	Sugerida ARIES	3.652	(69%)	3.336	(77%)	134	(60%)	3.336	(77%)	8.273	(82%)	7.362	(35%)	164	(226%)
	Original	11.950	**Dif.**	9.508	**Dif.**	270	**Dif.**	9.508	**Dif.**	29.040	**Dif.**	17.045	**Dif.**	452	**Dif.**
S29	Sugerida SRT	4.751	-1.531	2.359	-2.445	106	-93	2.359	-2.445	6.224	-10.230	9.329	-560	1.069	-900
	Sugerida ARIES	6.282	(24%)	4.804	(51%)	199	(47%)	4.804	(51%)	16.454	(62%)	9.889	(6%)	169	(533%)
	Original	402	**Dif.**	356	**Dif.**	98	**Dif.**	356	**Dif.**	533	**Dif.**	1.264	**Dif.**	81	**Dif.**
S31	Sugerida SRT	216	-186	169	-187	67	-31	169	-187	230	-303	1.264	-186	133	-52
	Sugerida ARIES	402	(46%)	356	(53%)	98	(32%)	356	(53%)	533	(57%)	1.078	(15%)	81	(64%)
	Original	1.519	**Dif.**	1.063	**Dif.**	110	**Dif.**	1.063	**Dif.**	1.749	**Dif.**	2.437	**Dif.**	175	**Dif.**
S32	Sugerida SRT	1.257	-262	861	-202	65	-45	861	-202	1.282	-467	2.238	-199	220	-45
	Sugerida ARIES	1.519	(17%)	1.063	(19%)	110	(41%)	1.063	(19%)	1.749	(27%)	2.437	(8%)	175	(26%)
	Original	9.390	**Dif.**	4.825	**Dif.**	721	**Dif.**	4.825	**Dif.**	9.198	**Dif.**	8.277	**Dif.**	288	**Dif.**
S36	Sugerida SRT	4.908	-2.964	1.803	-2.247	589	-106	1.803	-2.247	2.839	-4.929	5.255	-1.846	893	-618
	Sugerida ARIES	7.872	(38%)	4.050	(55%)	695	(15%)	4.050	(55%)	7.768	(63%)	7.101	(26%)	275	(225%)

Tabela 8.8 - Estados Original e Sugerido por SRT e por ARIES e Diferença entre esses Estados dos Sistemas de Software Analisados (Conclusão)

#	Estrutura	Ca		Ce		$LCOM4_p$		CBO_p		MPC_p		RFC_p		TCC_p	
	Original	4.694	**Dif.**	4.550	**Dif.**	273	**Dif.**	4.550	**Dif.**	7.226	**Dif.**	8.910	**Dif.**	87	**Dif.**
S38	Sugerida SRT	555	-4.139	400	-4.150	124	-149	400	-4150	579	-664	4.793	-4.117	533	-446
	Sugerida ARIES	4.694	(88%)	4.550	(91%)	273	(55%)	4.550	(91%)	7.226	7(92%)	8.910	(46%)	87	(513%)
	Original	11	**Dif.**	7	**Dif.**	16	**Dif.**	7	**Dif.**	16	**Dif.**	96	**Dif.**	9	**Dif.**
S40	Sugerida SRT	7	-4	3	-4	16	0	3	-4	6	-10	92	-4	11	-2
	Sugerida ARIES	11	(36%)	7	(57%)	16	(0%)	7	(57%)	16	(63%)	96	(4%)	9	(22%)
	Original	24.540	**Dif.**	16.679	**Dif.**	1.318	**Dif.**	16.679	**Dif.**	23.679	**Dif.**	25.974	**Dif.**	1.559	**Dif.**
S42	Sugerida SRT	12.749	-7.701	5.133	-8.649	861	-365	5.133	-8.647	8.039	-11.755	14.625	-7.142	2.841	-1.888
	Sugerida ARIES	20.450	(38%)	13.782	(63%)	1.226	(30%)	13.780	(63%)	19.794	(59%)	21.767	(33%)	952	(198%)
	Original	18	**Dif.**	18	**Dif.**	13	**Dif.**	18	**Dif.**	104	**Dif.**	233	**Dif.**	4	**Dif.**
S44	Sugerida SRT	9	-9	9	-9	9	-4	9	-9	40	-64	224	-9	6	-2
	Sugerida ARIES	18	(50%)	18	(50%)	13	(31%)	18	(50%)	104	(62%)	233	(4%)	4	(50%)
	Original	1.245	**Dif.**	1.242	**Dif.**	449	**Dif.**	1.242	**Dif.**	2.222	**Dif.**	4.651	**Dif.**	63	**Dif.**
S47	Sugerida SRT	536	-709	533	-709	338	-111	533	-709	895	-1.327	3.946	-705	181	-117
	Sugerida ARIES	1.245	(57%)	1.242	(57%)	449	(25%)	1.242	(57%)	2.222	(60%)	4.651	(15%)	63	(187%)
	Original	180	**Dif.**	180	**Dif.**	23	**Dif.**	180	**Dif.**	304	**Dif.**	699	**Dif.**	25	**Dif.**
S49	Sugerida SRT	10	-170	10	-170	13	-10	10	-170	18	-286	529	-170	34	-9
	Sugerida ARIES	180	(94%)	180	(94%)	23	(43%)	180	(94%)	304	(94%)	699	(24%)	25	(36%)
	Original	216	**Dif.**	159	**Dif.**	209	**Dif.**	159	**Dif.**	386	**Dif.**	2.354	**Dif.**	77	**Dif.**
S50	Sugerida SRT	153	-63	57	-102	191	-18	57	-102	94	-292	2.252	-102	102	-24,7
	Sugerida ARIES	216	(29%)	159	(64%)	209	(9%)	159	(64%)	386	(76%)	2.354	(4%)	77	(31%)

Fonte: Do autor (2017). Legenda: “Dif.” Refere-se a abreviatura de diferença

Tabela 8.9 - Estatística Descritiva da Avaliação Externa

Medidas Estatísticas	Variáveis Analisadas						
	Ca	**Ce**	**$LCOM4_p$**	**CBO_p**	**MPC_p**	**RFC_p**	**TCC_p**
Mínimo	-9.552	-9.400	-1.180	-9.398	-12.898	-17.970	-1.888,51
Máximo	223	177	297	177	180	560	465,4424
1º Quartil	-2.809,5	-2.764,5	-143	-2.764,5	-5.656,5	-2.504	-608,786
3º Quartil	-80,5	-95	-10	-95	-289	-64	-17,1033
Média	-2.062,11	-2.101,67	-122,815	-2.100,04	-3.484,7	-2.140,04	-359,144
Mediana	-709	-709	-49	-709	-1.327	-415	-117,453
Soma	-55.677	-56.745	-3.316	-56.701	-94.087	-57.781	-9.696,88
Erro padrão da média	517,3134	528,5571	48,35146	528,7025	780,5473	740,589	94,5988
Variância	7.225.556	7.543.061	63.122	7.547.210	16.449.862	14.808.745	241.621
Desvio Padrão	2.688,04	2.746,463	251,2415	2.747,219	4.055,843	3.848,213	491,5498
Assimetria	-1,31084	-1,33928	-2,65406	-1,33797	-0,89351	-2,6828	-1,19461
Curtose	0,70744	0,762434	9,038456	0,758876	-0,49312	7,905474	1,480509

Fonte: Do autor (2017).

A medida Soma apresentou valor negativo em todas as variáveis analisadas, revelando que os aprimoramentos proporcionados por SRT na estrutura dos sistemas de software analisados foram melhores do que os proporcionados por ARIES, pois o resultado, a diferença entre as estruturas sugeridas (`estrutura sugeria por ARIES – estrutura sugerida por SRT`), apresentou valor negativo.

A medida Erro Padrão da Média indica que, as medidas $LCOM4_p$ e TCC_p são as variáveis com menor valor para essa medida; assim, o valor da medida Média dessas medidas é mais representativo da população que o valor da medida Média das demais variáveis. Dessa maneira, se uma nova amostra de sistemas de software for coletada, processada por essas duas ferramentas e a diferença for analisada, as variáveis com maior chance de possuírem valor médio próximo ao da amostra trabalhada nessa análise são as medidas $LCOM4_p$ e TCC_p.

A medida Variância indica que as variáveis analisadas na estatística descritiva apresentam alto valor para essa medida. Isso pode ser compreendido como ampla distância entre o valor médio e o valor obtido pelos elementos da amostra analisada. O alto valor da medida Variância pode ser explicado pela variedade de tamanho dos sistemas analisados que pode ocasionar a existência de valores altos e baixos em função do tamanho do sistema e não em função da medida analisada. Esse alto valor também é percebido na medida Desvio Padrão, revelando que os valores dos elementos das variáveis analisadas estão dispersos em relação à medida Média e que o valor apresentado por essa medida não é confiável. O alto valor da medida Desvio Padrão pode ser explicado pela variabilidade de tamanho dos sistemas analisados, que leva a obtenção de valores dispersos em relação a medida Média. A medida

Assimetria indica que as variáveis analisadas possuem assimetria negativa, assim suas distribuições sugerem concentração de valores à direita da medida Média amostral. Isso é um indicativo que os dados não possuem uma distribuição normal.

A medida Curtose indica que as medidas $LCOM4_p$ e RFC_p possuem distribuição de frequência leptocúrtica, pois possuem o valor da medida Curtose maior que 3. Portanto, seus valores estão mais concentrados junto à média amostral, formando uma curva mais "aguda". As demais variáveis analisadas apresentam distribuição platicúrtica, pois possuem a medida Curtose menor que 3, possuindo valores mais dispersos e a curva formada é mais "achatada". Portanto, a medida Curtose dessas variáveis reforça o indicado pelas medidas Assimetria, Mediana e Média: as variáveis analisadas não possuem distribuição normal.

A estatística descritiva indica que as variáveis de forma independente não seguem distribuição normal. Entretanto, para verificar se as variáveis utilizadas de forma correlacionada seguem distribuição normal, um teste de normalidade multivariado foi realizado. Assim como na avaliação interna, foi utilizado o teste *E-test* para verificar a normalidade multivariada dos dados. Esse teste possui duas hipóteses:

H_0 = os dados seguem distribuição normal

H_1 = os dados não seguem distribuição normal

O resultado desse teste de normalidade (Figura 8.6), utilizando os dados da Tabela 8.8, indica, com confiabilidade de 95%, que os dados trabalhados não seguem distribuição normal multivariada, pois o p-valor (2×10^{-16}) obtido com o teste é menor que o nível de significância (5%). Portanto, rejeita-se H_0 e aceita-se H_1.

Figura 8.6 - Resultado do Teste de Normalidade dos Dados da Avaliação Externa

```
Energy test of multivariate normality: estimated parameters

data:  x, sample size 27, dimension 7, replicates 999
E-statistic = 7, p-value <0.0000000000000002
```

Fonte: Do autor (2017).

Para verificar se o tratamento aplicado por SRT foi superior ao tratamento aplicado por ARIES, foi utilizado um teste de médias, cujo objetivo é verificar se a média dos valores das medidas apresentada na estrutura sugerida por SRT foi "melhor" do que a média das medidas apresentada na estrutura sugerida por ARIES. Para realizar esse teste considerando as variáveis analisadas e a correlação existente entre elas, foi empregado o teste estatístico multivariado *T2 de Hotelling*. Esse teste foi realizado de maneira unilateral seguindo as seguintes hipóteses:

$$H_0 = \mu_d = \delta_0$$

$$H_1 = \mu_d > \delta_0$$

sendo H_0 a hipótese nula, que indica a inexistência de diferença entre as médias das medidas apresentadas pelas estruturas sugeridas das ferramentas comparadas, e H_1 a hipótese alternativa, que indica se a média dos valores das medidas nas estruturas sugeridas por SRT é superior à média das medidas apresentada na estrutura sugerida por ARIES.

Para aplicar o teste *T2 de Hotelling*, foi utilizada a função `HotellingsT2Test` do pacote `DescTolls` do software `R`. O resultado desse teste indica que H_0 deve ser rejeitada, pois o p-valor para o teste é igual a $1,0 \times 10^{-4}$ (Figura 8.7), sendo menor que o nível de significância (5%). Assim, com 95% de confiabilidade, há evidências significativas de que o efeito da reestruturação aplicada por SRT é mais significativo que o efeito proporcionado por ARIES sobre as medidas de software analisadas. O valor desse teste foi ratificado com a função `hotelling.test` do pacote `Hotelling`, que também aplica o teste *T2 de Hotelling* e apresentou o mesmo resultado.

Figura 8.7 - Resultado do Teste T2 de Hotelling sobre os Dados da Avaliação Externa

```
Hotelling's two sample T2-test

data:  dadosSRT and dadosARIES
T.2 = 4, df1 = 7, df2 = 50, p-value = 0.001
alternative hypothesis: true location difference is not equal to c(0,0,0,0,0,0,0)
```

Fonte: Do autor (2017).

Visto que existe diferença significativa entre as estruturas sugeridas por SRT e por ARIES, então é importante saber quais das variáveis analisadas (medidas de software) foram responsáveis por essa diferença. Isso pois, o teste de médias aplicado apresenta resultado geral, considerando a correlação entre as variáveis e impedindo que se saiba se houve alguma variável não responsável pela diferença entre as médias. Para isso, foram construídos intervalos de confiança simultâneos de forma semelhantes ao realizado na avaliação interna. O resultado da construção desses intervalos (Tabela 8.10) revela os valores mínimo e máximo desses intervalos para cada variável trabalhada.

Para verificar nos intervalos de confiança simultâneos construídos se uma das variáveis analisadas foi responsável pela rejeição da hipótese nula, deve-se verificar se as coordenadas do ponto $[0,0,0,0,0,0,0]^T$ estão contidas em todos os intervalos da variável. Em caso positivo, essa variável aceita H_0. Caso contrário, se pelo menos um dos intervalos não conter a coordenada 0, então essa variável é responsável pela rejeição de H_0. A análise do resultado da

construção desses intervalos indica que as sete variáveis analisadas são responsáveis pela rejeição H_0, pois, pelo menos, um ponto não contém a coordenada 0. Assim, há evidências significativas que os valores das sete medidas de software utilizadas foram melhores na estrutura sugerida por SRT do que os valores apresentados pelas medidas na estrutura sugerida por ARIES.

Tabela 8.10 - Intervalos de Confiança dos Dados da Avaliação Externa

Variáveis Analisadas	Intervalos	Variáveis Analisadas						
		Ca	**Ce**	**$LCOM4_p$**	**CBO_p**	**MPC_p**	**RFC_p**	**TCC_p**
Ca	Mínimo	-4.464	-4.483	-2.464	-4.483	-4.874	-4.817	-2.739
	Máximo	340	359	-1.660	359	750	693	-1.385
Ce	Mínimo	-4.523	-4.557	-2.506	-4.557	-4.980	-4.857	-2.823
	Máximo	319	353	-1.698	353	776	653	-1.381
$LCOM4_p$	Mínimo	-525	-527	-378	-527	-551	-295	-334
	Máximo	279	281	132	281	305	49	88
CBO_p	Mínimo	-4.521	-4.555	-2.504	-4.555	-4.979	-4.855	-2.821
	Máximo	321	355	-1.696	355	779	655	-1.379
MPC_p	Mínimo	-6.657	-6.723	-4.273	-6.724	-7.470	-6.962	-4.755
	Máximo	-1.033	-967	-3.417	-966	-220	-728	-2.935
RFC_p	Mínimo	-4.895	-4.895	-2.312	-4.895	-5.257	-5.579	-2.538
	Máximo	615	615	-1.968	615	977	1.299	-1.742
TCC_p	Mínimo	-1.036	-1.080	-570	-1.080	-1.269	-757	-798
	Máximo	318	362	-148	362	551	39	80
Resultado		**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**	**Rejeita H_0**

Fonte: Do autor (2017).

8.5 Discussão dos resultados

Nesta seção, são apresentadas discussões dos resultados obtidos nas avaliações interna e externa.

8.5.1 Resultados da avaliação interna

Com base nos dados obtidos do processo de reestruturação e nos testes estatísticos executados na avaliação interna, pode-se inferir que ARS atinge seu objetivo, pois foi comprovado estatisticamente que as sete medidas de software utilizadas apresentam melhoria significativa. Além disso, a quantidade de dependências entre pacotes foi reduzida em todos os

casos analisados. Portanto, essa abordagem aperfeiçoa simultaneamente os atributos acoplamento e coesão, sugerindo um software mais modularizado e manutenível.

Essa melhoria proporcionada às medidas de software e às dependências entre pacotes deve-se ao princípio básico dessa abordagem, mover classes entre pacotes. Esse princípio é reforçado pela forma como ARS determina a classe a ser movimentada (`PMov`) que prioriza o movimento da classe que gere a "maior perda" de acoplamento e "maior ganho" de coesão. Assim, busca-se reduzir a quantidade de dependências que as classes possuem entre os pacotes. Consequentemente, alcança-se um sistema de software mais modularizado, pois, reduzindo as dependências entre pacotes, são aumentadas as dependências dentro dos pacotes, gerando pacotes menos acoplados e mais coesos.

Além disso, ARS não tem seu processamento influenciado pelo tamanho do sistema de software reestruturado, pois essa abordagem age sobre as oportunidades de reestruturação existentes, classes que possuem mais dependências externas ao seu pacote do que internas. Portanto, o sistema de software foi aprimorado em função da quantidade de classes mal posicionadas na estrutura, de acordo com essa abordagem, e não em relação ao seu tamanho.

Entretanto, nas reestruturações executadas, alguns sistemas de software apresentaram mais de 60% das classes movimentadas (Figura 8.3). Esse fato pode ser um empecilho para o mantenedor compreender a estrutura sugerida, pois existem muitas diferenças em relação à estrutura original. Existem dois motivos que podem justificar essa quantidade de classes movimentadas:

a) As manutenções realizadas no sistema podem ter aumentado a quantidade de classes mal posicionadas na sua estrutura. O alto percentual de classes movimentadas se deve a sua má organização estrutural, que exige vários movimentos para aprimorá-la;
b) Não aplicar restrições de movimentações de classes entre os pacotes durante a reestruturação. Alguns movimentos de ARS podem ser inadequados gerando violações arquiteturais, como movimentar classes entre pacotes da camada de *Model* e *Controller*.

Ainda em relação as modificações estruturais que dificultam a compreensão da estrutura sugerida para o mantenedor, ARS não modifica a estrutura de pacotes do sistema, não dividindo ou unindo pacotes. Assim, o impacto gerado à estrutura dos sistemas de software será menor, facilitando a compreensão do mantenedor, visto que somente altera-se a estrutura de classes, mantendo intacta a estrutura de pacotes. Isso se deve ao princípio de ARS, que somente movimenta classes entre pacotes, e à heurística de movimentação `HMI`, que evita movimentar

a (única) classe de um pacote para outro pacote. Desse modo, é garantido que a estrutura de pacotes sugerida seja idêntica à estrutura original, facilitando a compreensão do mantenedor.

8.5.2 Resultados da avaliação externa

Os dados e as análises executados na avaliação externa indicam que a reestruturação realizada por SRT teve melhor desempenho, pois proporcionou melhoria estatisticamente mais significativa às medidas de software analisadas e, consequentemente, melhor estrutura sugerida do que a sugerida por ARIES. Esse melhor desempenho pode ser justificado pela abordagem de reestruturação utilizada por SRT (movimentação de classes entre pacotes) que pode ser mais eficaz que a abordagem utilizada por ARIES (divisão de pacotes).

A "melhor" eficácia de SRT pode ser reflexo da maior quantidade de oportunidades de reestruturações pelo movimento de classes entre pacotes do que pela divisão de pacotes, pois, para movimentar uma classe entre pacotes, é necessário que a classe possua mais relacionamentos com o pacote destino do que seu atual. Por outro lado, para sugerir a divisão de um pacote, é necessário que as classes que formarão o novo pacote não possuam dependências com demais classes do pacote. Essa menor quantidade de oportunidades para a abordagem de divisão de pacotes pode ser comprovada pelo fato que somente 7 dos 27 sistemas de software analisados (S5, S7, S17, S25, S29, S36 e S42 - Tabela 8.8), representando aproximadamente 26% do total, receberam sugestão de divisão de pacotes por ARIES. SRT realizou o movimento de classes entre pacotes em 100% dos sistemas de software analisados. Além disso, a "maior" eficácia de SRT pode ser observada nas situações em que as duas ferramentas conseguiram aplicar suas reestruturações sobre os sistemas de software. As melhorias proporcionadas por SRT foram superiores às melhorias proporcionadas por ARIES, para todas as medidas analisadas.

Outro fator crucial para essa maior eficácia de ARS é o método de determinação da classe a ser movimentada (PMov), pois busca movimentar a classe que proporcione maior ganho de coesão e maior perda de acoplamento a cada iteração. Desse modo "boas" oportunidades de reestruturação são encontradas para aprimorar a estrutura do sistema de software, aumentando sua coesão e reduzindo seu acoplamento. Outro ponto a ser analisado entre essas ferramentas é a geração de novos pacotes na estrutura do software. Em ARIES, novos pacotes são inseridos, possibilitando ao mantenedor determinar seus nomes antes de finalizar a reestruturação. A reestruturação efetuada por SRT não insere, exclui ou divide

pacotes na estrutura do software, o que pode facilitar a compreensão da estrutura sugerida após a reestruturação, pois a estrutura mantém-se inalterada em relação à estrutura original. Além disso, as classes movimentadas na estrutura sugerida estão mapeadas em um *log* de movimentos disponibilizado ao mantenedor para ter conhecimento e poder atualizar a documentação do sistema. Um ponto positivo em ambas as ferramentas consiste na solicitação de aprovação das alterações sugeridas antes de persisti-las no *workspace* da plataforma Eclipse IDE.

O tempo gasto pelas ferramentas para realizar a reestruturação não pode ser comparado ou discutido, pois ARIES não o disponibiliza. Uma solução seria determiná-lo por meio da medição utilizando uma terceira ferramenta, mas isso poderia incorrer em erros de precisão na medição do tempo, prejudicando a comparação.

8.6 Considerações finais

Neste capítulo, foi apresentada a avaliação de eficiência de ARS. Para realizar essa avaliação, foi utilizado um conjunto de 50 sistemas de software coletados dos repositórios `Git` e `Sourceforge`. Com base nesses sistemas duas avaliações foram realizadas.

Na primeira avaliação, esses sistemas foram reestruturados com SRT, *plug-in* que automatiza ARS, coletando dados das estruturas de cada sistema antes e depois da reestruturação. Com base nesses dados, análises estatísticas foram realizadas para determinar se a estrutura sugerida é melhor do que a original. Os resultados obtidos indicam a existência de evidências significativas que os sistemas de software reestruturados foram aprimorados, tornando-se menos acoplados e mais coesos (mais modulares e mais manuteníveis), pois as sete medidas de software utilizadas melhoraram significativamente e a quantidade de dependências entre pacotes foi reduzida entre as estruturas original e sugerida.

Na segunda avaliação, ARS foi comparada com uma abordagem semelhante. Para isso, foi feita uma busca por ferramentas que aplicam abordagens semelhantes, encontrando somente uma ferramenta disponível para *download* e funcionando adequadamente, ARIES. Com base nessa ferramenta e em SRT, um estudo comparativo da eficácia foi realizado. Nesse estudo, 27 sistemas de software foram reestruturados pelas ferramentas e suas estruturas sugeridas foram comparadas em relação as medidas de software utilizadas.

Os resultados dessa comparação indicam que a estrutura sugerida por SRT é melhor do que a sugerida por ARIES, pois foi comprovado estatisticamente que as sete medidas de software analisadas apresentam valores melhores. Esse melhor desempenho foi atribuído a menor granularidade de atuação da abordagem proposta em face à abordagem comparada, pois

existem mais oportunidades de reestruturações movimentando classes entre pacotes do que dividindo pacotes. Outro fator que influenciou nesse “melhor” desempenho foi a forma de determinar a probabilidade de movimentação de classes entre pacotes em SRT, encontrando “boas” oportunidades de reestruturação e gerando mais ganhos à estrutura do sistema de software.

9 CONSIDERAÇOES FINAIS

Neste capitulo, são apresentadas as conclusões, contribuições, lições aprendidas, ameaças à validade, limitações, trabalhos futuros e resultados de publicações desta investigação

9.1 Conclusões

Neste trabalho, foi apresentada uma abordagem para reestruturar sistemas de software orientados a objetos (ARS) por meio da movimentação de classes entre pacotes. O objetivo foi gerar sistemas de software com melhor estrutura interna (mais coesos e menos acoplados), tornando-os mais modulares e manuteníveis, pois ao torná-los mais modulares, consequentemente, tornam-se mais manuteníveis, pois a modularidade é uma subcaracterística da manutenibilidade. Para atingir esse objetivo, ARS apresenta um conjunto de passos para realizar a reestruturação, utilizando medidas de software para determinar o estado do sistema de software, e heurísticas para tomar decisões ao longo da reestruturação. Essa abordagem é apoiada pela heurística *Simulated Annealing* para evitar a subjetividade da avaliação humana e os mínimos locais no espaço de busca, aumentando as chances de sucesso da utilização da abordagem.

A avaliação de ARS revela sua capacidade de sugerir sistemas de software com melhor qualidade interna (menos acoplado, mais coeso, mais modular e mais manutenível), pois há evidências estatisticamente significativas que o valor das medidas de software utilizadas (Ca, Ce, CBO_p, RFC_p, MPC_p, TCC_p e $LCOM4_p$) foram aprimoradas entre as estruturas internas original e sugerida. Com ARS, houve redução da quantidade de dependências entre pacotes nos casos analisados, ratificando o aperfeiçoamento na modularidade. Outros pontos positivos da abordagem proposta são:

a) Não alterar a estrutura de pacotes do sistema de software reestruturado;
b) Aprimorar o acoplamento e a coesão de sistema de software;
c) Aperfeiçoar sistemas de software em função do quão deteriorado está sua organização estrutural e não em função do seu tamanho.

ARS foi comparada com uma abordagem semelhante existente na literatura. Nessa comparação, foram detectadas evidências significativas que a estrutura sugerida por ARS é melhor do que a estrutura sugerida pela abordagem existente na literatura, pois ARS apresenta melhores valores para as sete medidas de software utilizadas. Esse “melhor” desempenho foi

creditado à técnica de reestruturação utilizada (movimentação de classes entre pacotes e a forma de determinação da classe a ser movimentada), que possui menor granularidade de atuação do que a abordagem comparada e consegue encontrar mais oportunidades de reestruturação, gerando mais ganhos.

Esses resultados indicam que ARS recupera ou melhora a qualidade interna de sistemas de software, tornando-os mais modulares e manuteníveis. Desse modo, ARS age na contracorrente da degradação da qualidade ocasionada pelas manutenções mal planejadas. Isso possibilita que futuros processos de manutenção sejam realizados com menos dificuldade, reduzindo os recursos e os esforços empregados na condução desse processo. Portanto, há indícios que ARS pode propiciar ganhos à qualidade de sistemas de software.

9.2 Contribuições

A condução desta pesquisa contribuiu para a evolução do conhecimento na área de Ciência da Computação, mais especificamente na área de Engenharia de Software. As principais contribuições proporcionadas foram:

a) Uma Revisão Sistemática da Literatura, que identifica medidas de qualidade externa de software e atributos e medidas que as impactam;
b) Uma abordagem de reestruturação de sistemas de software orientados a objetos, para agir na contracorrente da degradação da qualidade, aprimorando a qualidade interna de sistemas de software;
c) Proposição de novas medidas de software para avaliar o ganho proveniente do movimento de classes entre pacotes;
d) Heurísticas para determinar (i) a probabilidade movimentação de classes entre pacotes, (ii) a melhor classe a ser movimentada na estrutura do software e (iii) a aceitação da estrutura sugerida do software;
e) A adaptação da heurística *Simulated Annealing* ao contexto de reestruturação de sistemas de software;
f) Uma ferramenta, *plug-in* para a plataforma Eclipse IDE (SRT - *Software Restructuring Tool*), que automatiza ARS (Abordagem para Reestruturação de Software).

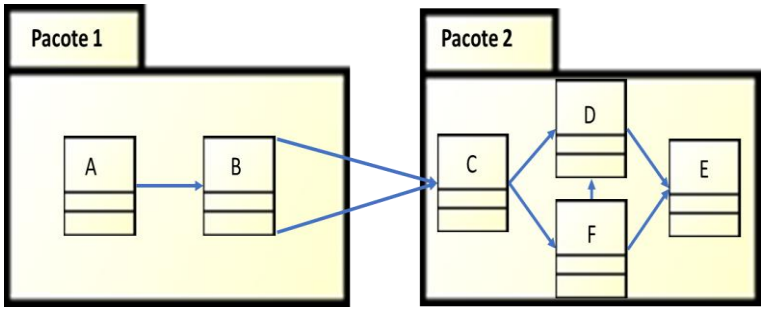
9.3 Lições aprendidas

Em versões iniciais de ARS, a medida LCOM (CHIDAMBER; KEMERER, 1991) foi utilizada, mas, em análises para verificar se essa abordagem atinge o objetivo desejado, foi detectado que, em alguns casos, essa medida não era aprimorada pela 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 medida LCOM foram aprofundados. Por fim, foi constatado que essa medida não repercute adequadamente os aprimoramentos proporcionados ao sistema de 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 sistema de software caso a quantidade de classes existentes no pacote destino seja maior do que a quantidade de classes que se relacionam com a classe movimentada.

Para facilitar a compreensão, no exemplo da Figura 9.1, ao movimentar a classe `B` do pacote `Pacote1` para o pacote `Pacote2`, sua coesão e a coesão da classe `C` aumentaram, pois elas possuem dependências em comum. Porém, a coesão das classes `D`, `E` e `F` reduziram (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 sistemas de software não trilhem o mesmo caminho, evitando abordagens infrutíferas e perda de recursos.

Figura 9.1 - Exemplo de Deterioração com a Medida LCOM



Fonte: Do autor (2017).

9.4 Ameaças à validade

Por mais criterioso que seja o planejamento e o desenvolvimento empregados na condução de uma pesquisa, existem questões que podem impactar nos resultados obtidos. A essas questões, dá-se o nome ameaças à validade e podem ser classificadas em quatro tipos (TRAVASSOS; GUROV; AMARAL, 2002):

a) **Validade interna** define se a relação entre o tratamento e o resultado é causal e não foi influenciada por outro fator não controlado ou não medido durante o estudo. Nesta investigação, acredita-se que a relação entre o tratamento (reestruturação do sistema de software) e o resultado (aprimoramento da estrutura interna do sistema de software) é causal, pois foi estatisticamente verificado que existe relação de causa e efeito entre o processo de reestruturação e a melhoria das medidas de software, que avaliam diretamente a estrutura interna do sistema de software;
b) **Validade externa** define a habilidade de generalizar os resultados do tratamento feito no experimento para o ambiente industrial, generalizando os resultados a outros contextos. Os problemas voltados à validade externa normalmente são a população de participantes não ser representativa da população de interesse, a instrumentação utilizada não condizer com a empregada em um ambiente real de produção e a outras condições específicas do ambiente de experimento não refletirem o ambiente de produção. Esses e outros problemas impedem a generalização dos resultados. Na investigação realizada, o tratamento foi executado sobre um conjunto de sistemas de software tido como representativo da população de interesse. As condições do ambiente e a instrumentação empregada nos experimentos executados são semelhantes às condições de um ambiente de produção, pois foi empregado a plataforma Eclipse IDE para dar suporte ao *plug-in* desenvolvido. Essa semelhança de condições proporciona maior robustez aos resultados, os quais podem ser facilmente replicados. Entretanto, uma premissa dessa pesquisa foi propor uma abordagem para reestruturação de sistemas de software orientados a objetos, porém na avaliação conduzida somente sistemas de software desenvolvidos na linguagem Java foram analisados. Isso representa uma ameaça à validade externa deste trabalho, pois existem outras linguagens orientadas a objetos. Outro ponto que configurar-se como uma ameaça à validade externa é a desconsideração da semântica dos pacotes durante a reestruturação. Por exemplo, ao movimentar classes de pacotes da camada de *View* para pacotes da camada de *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 (*e.g.* Engenheiro de Software) possui conhecimento sobre o sistema de software para indicar limitações de pacotes que não devem ser reestruturados em conjunto. Esse fato foi inviável nas reestruturações conduzidas, pois os engenheiros de software responsáveis pelo desenvolvimento dos sistemas analisados não foram encontrados para ajudar na reestruturação;

c) **Validade de construção** refere-se ao relacionamento entre a teoria e a observação, avaliando se o tratamento reflete bem a causa e o resultado reflete bem o efeito. Por isso, durante a avaliação da validade de construção, os fatores humanos envolvidos na pesquisa devem ser avaliados e controlados para esses fatores não interferirem nos resultados obtidos. Na investigação conduzida, o tratamento empregado não considerou fatores humanos, pois os dados trabalhados são a saída de um sistema de software, que realiza o processamento de maneira automatizada e independente de fatores humanos. Portanto, fatores humanos não interferiram no tratamento executado;

d) **Validade de conclusão** refere-se a chegar à conclusão adequada, observando o relacionamento entre o tratamento executado e os resultados obtidos. Para avaliar essa validade, o teste estatístico escolhido, o tamanho da amostra trabalhada, a confiabilidade das medidas utilizadas e a confiabilidade de implementação dos tratamentos devem ser considerados. Nesta investigação, foram empregados testes estatísticos robustos e com alta confiabilidade. Esses testes tiverem sua adequação ao contexto desta pesquisa, avaliada e comprovada por um especialista na área estatística, o qual também auxiliou na definição do tamanho da amostra. Em relação as medidas estatísticas utilizadas e a implementação dos tratamentos, esses foram adequadamente construídos e analisados sem influência dos pesquisadores para obter o resultado, analisando os resultados das medidas e dos testes de maneira ética e condizente com a realidade.

9.5 Limitações

Uma limitação refere-se ao uso de medidas estáticas em ARS, pois elas podem gerar vieses 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 durante a execução do sistema de software. 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, somente foram utilizadas medidas estáticas.

Outra limitação deste trabalho consiste em utilizar somente um tipo de reestruturação para aprimorar a estrutura do sistema de software, pois existem diversos tipos de reestruturação. Nesta investigação, foi empregada somente a reestruturação baseada na movimentação de classes entre pacotes, para evitar diferenças entre a estrutura original e a estrutura sugerida do sistema, pois quanto mais alterações efetuadas na estrutura, mais esforço é exigido do mantenedor para compreende-la.

A ligação tardia (*late binding*) configura-se como outra limitação deste trabalho. Essa limitação ocorre por causa da existência da possibilidade de classes não analisadas serem carregadas durante a execução. Isso pode ocasionar vieses na movimentação de classes entre pacotes pois, mesmo que seja lógico movimentar uma classe entre dois pacotes em um cenário desconsiderando a ligação tardia, em um cenário considerando tais ligações esse movimento pode não ocorrer ou vice-versa.

Outra limitação dessa abordagem é não realizar modificações no código do sistema de software reestruturado. Portanto, sistemas que possuem um código de baixa qualidade, continuarão dessa forma, pois a abordagem proposta somente consegue movimentar classes entre pacotes para aprimorar a sua modularidade.

9.6 Trabalhos futuros

Como trabalhos futuros, sugere-se:

a) Realizar estudos considerando medidas dinâmicas para determinar a movimentação de classes entre pacotes;
b) Realizar estudos considerando outras medidas estáticas de software, para avaliar se ARS aprimora outros atributos do sistema de software;
c) Expandir ARS para realizar reestruturações em diversas granularidades, como mover métodos entre classes e dividir/unir pacotes;

d) Desenvolver mecanismo para reestruturar sistemas de software considerando o padrão arquitetural em que foram desenvolvidos;
e) Realizar avaliação qualitativa dos resultados com especialistas, para conhecer a qualidade da estrutura sugerida;
f) Desenvolver abordagem de visualização para comparar as estruturas original e sugerida, ressaltando as diferenças existentes entre elas e facilitando a compreensão do mantenedor;
g) Aperfeiçoar ARS para reduzir os impactos gerados à estrutura do sistema de software.

9.7 Resultados de publicações

A proposta e os resultados preliminares desta pesquisa foram publicados ao longo do seu desenvolvimento em uma escola, uma conferência e em um simpósio:

a) SANTOS D. B.; RESENDE A. M. P.; COSTA H. A. X. Melhoria da Qualidade Interna de Software Orientado a Objetos Utilizando Medidas de Acoplamento e Coesão. In: ESCOLA LATINO AMERICANA DE ENGENHARIA DE SOFTWARE, 2., 2015, Porto Alegre. **Anais**...Porto Alegre: UFRGS, 2015. p. 166-169.
b) 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, 42., 2016, Valparaíso. **Proceedings**… Valparaíso: [s. n.], 2016. p. 539-550.
c) SANTOS, D. B.; JUNIOR, P. A.; COSTA, H. Uma Abordagem Para Reestruturação de Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 15., 2016, Maceió. **Anais**...Maceió: [s. n.], 2016. p. 286-300.
d) SANTOS, D. B.; JUNIOR, P. A.; COSTA, H. SRT - A Tool for Restructuring Java Software. In: INTERNATIONAL CONFERENCE OF CHILEAN COMPUTER SCIENCE SOCIETY, 35., 2016, Valparaíso. **Proceedings**... Valparaíso: [s. n.], 2016. p. 259-266.

REFERÊNCIAS

AARTS, E.; KORST, J.; MICHIELS. W. **Local Search in Combinatorial Optimization**. 1. ed. New York: Ed. John Wiley & Sons, 1997.

ABDEEN, H.; DUCASSE, S.; SAHRAOUI, H.; ALLOUI, I. Automatic Package Coupling and Cycle Minimization. In: WORKING CONFERENCE REVERSE ENGINEERING, 16., 2009, Lille. **Proceedings**… Lille: IEEE, 2009. p. 103-112.

ABDEEN, H.; SAHRAOUI, H.; SHATA, O.; ANQUETIL, N.; DUCASSE, S. Towards Automatically Improving Package Structure While Respecting Original Design Decisions. In: WORKING CONFERENCE REVERSE ENGINEERING, 20., 2013, Koblenz-Landau. **Proceedings**… Koblenz-Landau: IEEE, 2013. p. 212-221.

ABREU, F. B. E.; CARAPUÇA, R. Object-Oriented Software Engineering: Measuring. In: INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, 4., 1994, McLean. **Proceedings**… McLean: ASQ, 2013. p. 1-8.

AGGARWAL, K. K.; SINGH, Y.; KAUR, A.; MALHOTRA, R. Empirical Study of Object-Oriented Metrics. **Journal of Object Technology**, Zurich, v. 5, n. 8, p. 149-173, Novembro de 2006.

AL DALLAL, J. Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics. **IEEE Transactions on Software Engineering**, Nova Jersey, v. 37, n. 6, p. 788-804, Novembro de 2011.

AL DALLAL, J. Object-Oriented Class Maintainability Prediction Using Internal Quality Attributes. **Information and Software Technology**, Newton, v. 55, n. 11, p. 2028-2048, Julho de 2013.

ARNOLD, R. S. Software Restructuring. **IEEE Transactions**, Herndon, v. 77, n. 4, p. 607-617, Abril de 1989.

BARBOSA, N.; HIRAMA, K. Assessment of Software Maintainability Evolution Using C&K Metrics. **IEEE Latin America Transactions**, New York, v. 11, n. 5, p. 1232-1237, Novembro de 2013.

BAVOTA, G. L.; MARCUS, A.; OLIVETO, R. Software Re-Modularization Based on Structural and Semantic Metrics. In: WORKING CONFERENCE REVERSE ENGINEERING, 17., 2010, Boston. **Proceedings**… Washington: IEEE Computer Society , 2010. p. 195-204.

BAVOTA, G. L.; MARCUS, A.; OLIVETO, R. Using Structural and Semantic Measures to Improve Software Modularization. **Empirical Software Engineering**, Detroit, v. 18, n. 5, p. 901-932, Outubro de 2013.

BAVOTA, G.; LUCIA A. D.; MARCUS, A.; OLIVETO, R.; PALOMBA, F. Supporting Extract Class Refactoring in Eclipse: The ARIES Project. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 34., 2012, Zurich. Proceedings… Zurich, 2012. p. 1419-1422.

BAVOTA, G. L.; GETHERS, M.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Improving Software Modularization Via Automated Analysis of Latent Topics and Dependencies. **ACM Transactions on Software Engineering and Methodology**, New York, v. 23, n. 2, p. 1-33, Fevereiro de 2014.

BIANCHI, A.; CAIVANO, D.; LANUBILE, F.; VISAGGIO, G. Evaluating Software Degradation Through Entropy. In: INTERNARIONAL SOFTWARE METRICS SOFTWARE SYMPOSIUM, 7., 2001, London. **Proceedings**…Washington: IEEE Computer Society, 2001. p. 210-219.

BIEMAN, J. M.; KANG, B. K. Cohesion and Reuse in an Object-Oriented System. **Software Engineering Notes ACM**, New York, v. 20, p. 259-262, Agosto de 1995.

BRYTON, S.; ABREU, F. B. Modularity-Oriented Refactoring. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 12., 2008, Athens. **Proceedings**… Washington: IEEE Computer Society, 2008. p. 294-297.

CHAE, H. S.; KWON, Y. R.; BAE, D. H. Cohesion Measure for Object-Oriented Classes. **Software - Practice and Experience**, New York, v. 30, n. 12, p. 1405-1431, Outubro de 2000.

CHARNES, A.; WOLFE, M. Extended Pincus Theorems and Convergence of Simulated Annealing. **International Journal of Systems Science**, Austin, v. 20, n. 8, p. 1521-1533, de 1989.

CHEN, Z.; ZHOU, Y.; XU, B.; ZHAO, J.; YANG, H. A Novel Approach to Measuring Class Cohesion Based on Dependence Analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 18., 2002, Montreal. **Proceedings**… Washington: IEEE Computer Society, 2002. p. 377-384.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **IEEE Transactions on Software Engineering**, Piscataway, v. 20, n. 6, p. 476-493, Junho de 1994.

CHIDAMBER, S. R.; KEMERER, C. F. Towards a Metrics Suite for Object-Oriented Design. **Object-Oriented Programming Systems, Languages and Applications**, New York, v. 26, n. 11, p. 197-211, Novembro de 1991.

COLENGHI, F. K. R.; MINGOTI, S. A. **Estudo Comparativo de Testes de Hipótese Multivariados para o Vetor de Médias via Simulação de Monte Carlo**. 2008. 134p. Dissertação (Mestrado em Estatística)–Universidade Federal de Minas Gerais, Belo Horizonte, 2008. Disponível em: < http://www.bibliotecadigital.ufmg.br/dspace/bitstream/handle/1843/RFFO-7UDM94/dissertacao_final_fernanda.pdf?sequence=1>

DAFFARA, C.; GONZÁLES-BARAHOMA, J. M.; HUMENBERGER, E.; KOCH, W. L., B.; LAURIE. B. **Free Software / Open Source: Information Society Opportunities for Europe?**. Versão 1.2, 2000. Disponível em: <http://eu.conecta.it/paper.pdf>. Acesso em: 15 Set. 2014.

DALGARNO, M. When Good Architecture Goes Bad. **Methods and Tools**, Boston, v. 17, n. 1, p. 27-34, 2009.

DU BOIS, B.; DEMEYER, S.; VERELST, J. Refactoring-Improving Coupling and Cohesion of Existing Code. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 11., 2004, Delft. **Proceedings**… Washington: IEEE Computer Society, 2004. p. 144-151.

DUCASSE, S.; ANQUETIL, N.; BHATTI, M. U.; HORA, A. C. S**oftware Metrics for Package Remodularization**. Relatório Técnico, Versão 1.0, 2011, 54p. Disponível em: < https://hal.inria.fr/hal-00646878v1> Acesso em: 25 Ago. 2015.

ECLIPSE. **Abstract Syntax Tree**. Disponível em: <https://www.eclipse.org/articles/article.php?file=ArticleJavaCodeManipulation_AST/index.html>. Acesso em: 29 Out. 2015.

ELISH, M. O. Exploring the Relationships Between Design Metrics and Package Understandability: A Case Study. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 18., 2010, Braga. **Proceedings**… Washington: IEEE Computer Society, 2010. p. 144-147. 2010.

ERDIL, K.; FINN, E.; KEATING, K.; MEATTLE, J.; PARK, S. **Software Maintenance as Part of the Software Life Cycle**. Relatório Técnico. 2003. 49p.

ERLIKH, L. Leveraging Legacy System Dollars for e-Business. **IT professional**, Piscataway, v. 2, n. 3, p. 17-23, Maio de 2000.

ERNST, M. D. Static and Dynamic Analysis: Synergy and Duality. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING Workshop on Dynamic Analysis, 25., 2003, Portland. **Proceedings**… New York: ACM, 2003. p. 24-27.

ESPINDOLA, R. S.; MAJDENBAUM, A.; AUDY, J. L. N. Uma Análise Crítica dos Desafios para Engenharia de Requisitos em Manutenção de Software. In: WORKSHOP EM ENGENHARIA DE REQUISITOS, 3., 2004, Tandil. **Anais**… Rio de Janeiro: Puc Rio, 2004. p. 226-238.

FENTON, N. E.; NEIL, M. Software Metrics: Roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, 22., 2000, Limerick. **Proceedings**… New York: ACM, 2000, p. 357-370.

FERREIRA, D. F. **Estatística Multivariada**. 2. ed. Lavras: Ed. UFLA, 2008.

FOCUS, M. **Measures and Metrics**. 2008. Disponível em: <http://supportline.microfocus.com/documentation/books/ev56/ev56books/acrobat/Measures%20and%20Metrics.PDF>. Acesso em: 13 Jun. 2015.

FOWLER, M. **Refactoring**: Improving the Design of Existing Code. 1. ed. Jyväskylä, Finland: Ed. Addison-Wesley, 1999.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**: Elements of Reusable Object-Oriented Software. 1. ed. Boston: Ed. Pearson Education, 1994.

GERHARDT, T. E.; SILVEIRA D. T. **Métodos de Pesquisa**. Porto Alegre: Ed. UFRGS, 2009.

GIL, A. C. **Como Elaborar Projetos de Pesquisa**. São Paulo: Ed. Atlas, 2002. 5 v.

GURP, V. J.; BOSH, J. Design Erosion: Problems and Causes. **Journal of Systems and Software**, New York, v. 61, n. 2, p. 105-119, Março de 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, 13., 2009, Kaiserslautern. **Proceedings**… Washington: IEEE Computer Society, 2009. p. 3-4.

HAIR, J. F.; BLACK, W. C.; BABIN, B. J.; ANDERSON, R. E.; TATHAM, R. L. **Análise Multivariada de Dados**. 6. ed. Porto Alegre: Ed. Bookman, 2009.

HITZ, M.; MONTAZERI, B. Measuring Coupling and Cohesion in Object-Oriented Systems. In: INTERNATIONAL SYMPOSIUM ON APPLID CORPORATE COMPUTING, 3., 1995, Monterrey. **Proceedings**…Monterrey: [s. n.], 1995. p. 1-10.

HONGLEI, T.; WEI, S.; YANAN, Z. The Research on Software Metrics and Software Complexity Metrics. In: INTERNATIONAL FORUM ON COMPUTER SCIENCE-TECHNOLOGY AND APPLICATIONS, 9., 2009, Chongqing. **Proceedings**… Washington: IEEE Computer Society, 2009. p. 131-136.

HOTELLING, Disponível em: <http://ncss.wpengine.netdna-cdn.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Hotellings_Two-Sample_T2.pdf>. Acesso em: 7 Jul. 2016.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **IEEE 610.12**: Standard Glossary of Software Engineering Terminology. 31 de Dezembro. 1990.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELETROTECHNICAL COMMISSION. **ISO/IEC 25000**. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation. 2014.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION / INTERNATIONAL ELETROTECHNICAL COMMISSIO. **ISO/IEC 25010**. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation - System and Software Quality Models. 2011.

JOHN, B.; KADADEVARAMATH, R. S.; IMMANUEL, E. A. Recent Advances in Software Quality Management: A Review. **Merit Research Journal of Business and Management**, Boston, v. 4, n. 3, p. 18-26, Junho de 2016.

JOSHI, P.; JOSHI, R. K. Quality Analysis of Object Oriented Cohesion Metrics. In: QUALITY OF INFORMATION AND COMMUNICATIONS TECHNOLOGY, 17., 2010, Oporto. **Proceedings**... Quality Analysis of Object Oriented Cohesion Metrics, 2010. p. 319-324.

JUNG, C. F. **Metodologia para Pesquisa & Desenvolvimento**: Aplicada a Novas Tecnologias, Produtos e Processos. 1. ed. Rio de Janeiro: Ed. Axcel Books, 2004.

KAN, S. H. **Metrics and Models in Software Quality Engineering**. 2. ed. Boston: Ed. Addison-Wesley, 2002.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. **Science**, New York, v. 220, n. 4598, p. 671-680, Maio de1983.

LANZA, M.; MARINESCU, R. **Object-Oriented Metrics in Practice**: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. 1. ed. Ed. Springer, 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, 4., 1995, Slovenia. **Proceedings**…Berlin: Springer, 1995. p. 81-90.

LEHMAN, M. Program, Life-Cycle, and the Law of Program Evolution. **IEEE**, New York, v. 68, n. 9, p. 1060-1076, Setembro de 1980.

LI, W.; HENRY, S. Object-Oriented Metrics that Predict Maintainability. **Journal of Systems and Software**, New York, v. 22, n. 2, p. 111-122, Novembro de 1993.

LIMA, E. D. C.; RESENDE A. M. P. **Uma Análise dos Valores de Referência de Algumas Medidas de Software**. 2014. 192 p. Dissertação (Mestrado em Ciência da Computação)– Universidade Federal de Lavras, Lavras. 2014.

LORENZ, M.; KIDD, J. **Object-Oriented Software Metrics**: a practical guide. 1. ed. Upper Saddle River: Ed. Prentice-Hal. 146p. 1994.

MAMONE, S. The IEEE Standard for Software Maintenance. **SIGSOFT Software Engineering Notes**, New York, v. 19, n. 1, p. 75-76, Janeiro de 1994.

MARTIN, R. OO Design Quality Metrics. An Analysis of Dependencies. In: WORKSHOP PRAGMATIC AND THEORETICAL DIRECTIONS IN OBJECT-ORIENTED SOFTWARE METRICS, 3., 1994, New York. **Proceedings**… New York: [s. n.], 1994. p. 151-170. 1994.

MOGHADAM, I. H.; CINNÉIDE, M. Code-Imp: A Tool for Automated Search-Based Refactoring. In: INTRENATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., 2011, Honolulu. **Proceedings**… Berlin: ACM, 2011. p. 41-44.

MURPHY, G. C.; MIK K.; LEAH, F. How are Java Software Developers using the Elipse IDE?. **Software IEEE**, Los Alamitos, v. 23, n. 4, p. 76-83, Julho de 2006.

NICOLAESCU, A.; LICHTER, H.; XU, Y. Evolution of Object Oriented Coupling Metrics: A Sampling of 25 Years of Research. In: INTERNATIONAL WORKSHOP ON SOFTWARE ARCHITECTURE AND METRICS, 15., 2015, Florence. **Proceedings**… Piscataway: IEEE, 2015. p. 48-54.

NIKOLOPOULOS, C. **Expert Systems**: Introduction to First and Second Generation and Hybrid Knowledge Based Systems. 1. ed. Peoria: Ed. Marcel Dekker, 1997.

OPDYKE, W.F. **Refactoring Object-Oriented Frameworks**. 1992. 206p. Tese (Doutorado de Filosofia em Ciência da Computação)–University of Illinois, Urbana, 1992.

PADUELLI, M. M.; SANCHES, R. Problemas em Manutenção de Software: Caracterização e Evolução. In: WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA, 3., 2006, Vila Velha. **Anais**... Vila Velha: [s. n.], 2006. p. 1-13.

PALOMBA, F.; TUFANO, M.; BAVOTA, G.; OLIVETO, R.; MARCUS, A.; POSHYVANYK, D.; DE LUCUA, A. Extract Package Refactoring in ARIES. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 37., 2015, Florença. **Proceedings**… Piscataway: IEEE, 2015. p. 669-672.

PARNAS, D. L. Software Aging. INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 16., 1994, Sorrento. **Proceedings**… Los Alamitos: IEEE Computer Society, 1994. p. 279-287.

PETERNELLI, L. A. **INF 162**. Disponível em: <http://www.dpi.ufv.br/~peternelli/inf162.www.16032004/> Acesso em: 19 Jun. 2016.

PFLEEGER, S. L.; ATLEE, J. M. **Software Engineering**: Theory and Practice. 4. ed. London: Pearson. 792p, 2009.

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, 13., 2014, Blumenau. **Proceedings**… Blumenau: [s. n.], 2014. p. 194-208.

POSHYVANYK, D.; MARCUS, A.; FERENC, R.; GYIM´OTHY, T. Using Information Retrieval Based Coupling Measures for Impact Analysis. **Empirical Software Engineering**, Hingham, v. 14, n. 1, p. 5-32, Fevereiro de 2009.

PRESSMAN, R.; MAXIM, B. **Software Engineering: A Practitioner's Approach**. 9. ed. Boston: McGraw-Hill, 2016.

RUBEL, D. The Heart of Eclipse. **Queue - System Evolution**, New York, v. 4, n. 8, p. 36-44, Outubro de 2006.

RUTENBAR, R. A. Simulated Annealing Algorithms: An Overview. **IEEE Circuits and Devices Magazine**, Rockville, v. 5, n. 1, p. 19-26, Janeiro de 1989.

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, 42, 2016a, Valparaíso. **Proceedings**… Valparaíso: [s. n.], 2016a p. 539-550.

SANTOS, D. B.; JUNIOR, P. A.; COSTA, H. Uma Abordagem Para Reestruturação de Sistemas de Software Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 15., 2016b, Maceió. **Anais**...Maceió: [s. n.], 2016b. p. 286-300.

SANTOS, D. B.; JUNIOR, P. A.; COSTA, H. SRT - A Tool for Restructuring Java Software. In: INTERNATIONAL CONFERENCE OF CHILEAN COMPUTER SCIENCE SOCIETY, 35, 2016c, Valparaíso. **Anais**... Valparaíso: [s. n.], 2016c. p. 259-266.

SARKAR, S.; RAMACHANDRAN, S.; KUMAR, G. S.; IYENGAR, M. K.; RANGARAJAN, K.; SIVAGNANAM, S. Modularization of a Large-Scale Business

Application: A Case Study. **IEEE Software**, Los Alamitos, v. 26, n. 2, p. 28-35. Março de 2009.

SCACCHI, W. Free/Open Source Software Development: Recent Research Results and Methods. **Advances in Computers**, Dubrovnik, v. 69, p. 243-295, Setembro de 2007.

SHELAJEV, O.; MAPLE, S. **Java Tools and Technologies Landscape Report 2016**. Disponível em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/> Acesso em: 17 Ago. 2016.

SILVA, L.; BALASUBRAMANIAM, D. Controlling Software Architecture Erosion: A Survey. **Journal of Systems and Software**, New York, v. 85, n. 1, p. 132-151, Janeiro de 2012

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, 41., 2015, Arequipa. **Proceedings**…Arequipa: [s. n.], 2015. p. 1-9.

SNEED, H. M.; BRÖSSLER, P. Critical Success Factors in Software Maintenance: A Case Study. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, 19., 2003, Amsterdam. **Proceedings**… Washington: IEEE Computer Society2003. p. 190-198.

SOMMERVILLE, I. **Software Engineering**. 10. ed. Boston: Ed. Addison-Wesley, 2015.

CASS S. **The 2015 Top Ten Programming Languages**. 2015. Disponível em: <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>. Acesso em: 2 Jun. 2016.

STEPHEN, E. G.; GRAVES, A. F.; MARRON, J. S.; MOCKUS, A. Does Code Decay? Assessing the Evidence from Change Management Data. **Transactions on Software Engineering**, Piscataway, v. 27, n. 1, p. 1-12, Janeiro de 2001.

STEVENS, W. P.; MYERS, G. J.; CONSTANTINE, L. L. Structured Design. **IBM Systems Journal**, Riverton, v. 13, n. 2, p. 115-139, Junho de 1974.

SUBRAMANYAM, R.; KRISHNAN, M. S. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. **IEEE Transactions on Software Engineering**, Piscataway, v. 29, n. 4, p. 297-310. 2003.

SWEBOK. **Guide to the Software Engineering Body of Knowledge**. IEEE Computer Society, 2014. v. 3.

SZÉKELY, G. J.; RIZZO, M. L. A New Test for Multivariate Normality. **Journal of Multivariate Analysis**, Orlando, v. 93, n 1. p. 58-80, Março de 2005.

TERRA, R.; VALENTE, M. T. Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. WORKSHOP DE MANUTENÇÃO DE SOFTWARE MOVERNA, 7., 2010, Belém. Proceedings...Belém: [s.n.], 2010.

TIOBE. **The Java Programming Language**. Disponível em: <http://www.tiobe.com/tiobe_index?page=Java>. Acesso em: 20 Jun. 2016.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. G. **Introdução à Engenharia de Software Experimental**. Relatório Técnico UFRJ. Rio de Janeiro, RJ, 2002.

VAN LAARHOVEN, P. J.; AARTS, E. H. **Simulated Annealing**: Theory and Applications. Norwell: Ed. Springer, 1987.

ZANETTI, M. S.; TESSONE, C. J.; SCHOLTES, I.; SCHWEITZER, F. Automated Software Remodularization Based on Move Refactoring. In: INTERNATIONAL CONFERENCE ON MODULARITY, 13., 2014, Lugano. **Proceedings**… New York: ACM, 2014. p. 73-84.