FERNANDA FLORIANO SILVA

DETECÇÃO PRECISA DE DIFERENÇAS RELACIONADAS A REFATORAÇÕES

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Leonardo Gresta Paulino Murta

Niterói

2013

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S586 Silva, Fernanda Floriano
Detecção precisa de diferenças relacionadas a refatorações / Fernanda Floriano Silva. – Niterói, RJ : [s.n.], 2013.
123 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2013.
Orientador: Leonardo Gresta Paulino Murta.

1. Engenharia de software. 2. Algoritmo diff. 3. Algoritmo húngaro. 4. Refatoração. I. Título.

CDD 005.1

FERNANDA FLORIANO SILVA

DETECÇÃO PRECISA DE DIFERENÇAS RELACIONADAS A REFATORAÇÕES

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Aprovada em Abril de 2013.

BANCA EXAMINADORA

_____

Prof. Dr. LEONARDO GRESTA PAULINO MURTA – Orientador
UFF - Universidade Federal Fluminense

_____

Prof. Dr. FÁBIO PROTTI
UFF - Universidade Federal Fluminense

_____

Prof. Dr. GUILHERME HORTA TRAVASSOS
COPPE / UFRJ - Universidade Federal do Rio de Janeiro

Niterói
2013

À minha família e amigos.

AGRADECIMENTOS

Dedico todo meu esforço na construção desta dissertação, inicialmente, aos tutores da minha vida. Ao meu pai Alvaro, mais conhecido como vovô Ovo, por ter me ensinado que um guerreiro nunca cansa e por toda sua dedicação como filho, pai e avô. A minha mãe Telma, a mulher maravilha, por ser minha eterna inspiração, meu exemplo. Aos dois, por todos os momentos dedicados a nossa família, por serem trabalhadores incansáveis e pessoas inigualáveis. Exemplos de vida e honestidade, sempre dispostos a ensinar o verdadeiro sentido da vida. Por me permitirem afirmar com toda segurança e certeza que eu tenho sim, os melhores pais do mundo.

A minha sobrinha Letícia, a Lele, por trazer serenidade e por me lembrar dos milagres da vida, com o seu nascimento durante a construção deste trabalho. Por cada sorriso dado e por me fazer imaginar como será gostoso acompanhar de perto o seu crescimento.

A minha sobrinha e afilhada Isabella, a Bellinha. Por ser minha grande amiga, minha princesinha, minha paixão. Por todas as vezes que ela

me ajudou a “fazer os trabalhos da escola” para que eu pudesse brincar de boneca com ela. Pelos momentos que ela sempre fez questão de estar ao meu lado e por me fazer sentir especial em sua vida.

A minha irmã mais velha Samantha, carinhosamente chamada de Samy, mãe da Lele e da Bellinha, pessoa pela qual possuo um amor infindável. Minha companheira de vida, meu apoio, muitas vezes inconsciente. A pessoa que me deu um presente maravilhoso, ser madrinha de sua primogênita Isabella.

Ao meu irmão Leonardo, o Leo, minha eterna saudade, meu anjo. Presente de Deus, que seguirá comigo durante toda minha vida, em meus pensamentos e em minhas orações.

Ao Lucas, meu companheiro, meu amigo, meu amor. Por compreender as ausências e me apoiar nos momentos de cansaço. Por não desistir da gente, apesar da rotina capaz de terminar com qualquer relacionamento.

Ao meu professor orientador Leonardo Murta, o Leo. Por todos os ensinamentos, paciência e compreensão. Por ser um professor como poucos, sempre me mantendo motivada a prosseguir com os estudos, sem esquecer os momentos importantes da vida.

Aos professores Guilherme Travassos e Fábio Protti por aceitarem participar desta banca.

Aos companheiros do GEMS, Grupo de Manutenção e Evolução de Software da UFF, comandado pelo professor Leo Murta, por todo conhecimento compartilhado.

Aos meus amigos de trabalho pelo constante apoio e incentivo, em especial Rafael Mansano e Rodolfo Wagner por terem iniciado comigo esta jornada.

A Deus por ter permitido a presença destas pessoas na minha vida. Junto delas foi bem mais fácil. Por ter me dado saúde para que eu pudesse atingir meus objetivos e principalmente por ter me dado sabedoria para lidar com a árdua rotina de intermináveis tarefas do Mestrado e horas diárias de trabalho. Tenho certeza que graças a essas pessoas, hoje sou uma pessoa melhor.

Cada ser em si carrega o dom de ser capaz, e ser feliz.

(Almir Sater)

RESUMO

Durante o ciclo de vida do software, artefatos mudam em resposta a demandas corretivas e evolutivas. Essas mudanças envolvem com frequência refatorações de código-fonte. A refatoração se concentra em melhorar os atributos de qualidade (requisitos não-funcionais), sem alterar o seu comportamento (requisitos funcionais). No entanto, os algoritmos convencionais de *diff* não capturam de forma precisa esse tipo de mudança, devido à falta ou ineficiência na detecção de movimentação de blocos de código, dentro e entre arquivos. Este trabalho apresenta uma abordagem, denominada IDIFF, destinada a apoiar o entendimento de como duas versões de software diferem, mesmo se mudanças relacionadas à refatoração forem realizadas. Para isso, são analisadas diferentes granularidades, a fim de detectar com precisão movimentos de blocos sem a necessidade de um poder computacional expressivo. Os resultados experimentais mostram que o IDIFF fornece resultados precisos ao utilizar diferentes granularidades (i.e., linha, palavra e caractere) e uma boa cobertura quando a granularidade linha é utilizada.

Palavras-chave: *diff*, refatoração, LCS, Algoritmo Húngaro.

ABSTRACT

During the software life cycle, artifacts change in response to both corrective and evolutive demands. These changes frequently entail source-code refactorings. Refactoring focuses on improving the software quality attributes (i.e., non-functional requirements) without changing its behavior (i.e., functional requirements). However, conventional diff algorithms do not precisely capture such type of change due to missing or inefficient detection of moving blocks of code within and among files. This work introduces an approach designed to support understanding how two software versions differ, even if refactoring-related changes were performed. It analyzes different granularities in order to precisely detect block moves without requiring expressive computational power. The experimental results show that IDIFF provides precise results for any granularity (i.e., line, word, and character) and good recall when the line granularity is used.

Keywords: diff, refactoring, LCS, Hungarian algorithm.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

DDiff: *Directory diff*

FDiff: *File diff*

IDIFF: *Iterative diff*

LCS: *Longest Common Subsequence*

SHA1: *Secure Hash Algorithm*

SUMÁRIO

Capítulo 1 – **INTRODUÇÃO**

1.1 **MOTIVAÇÃO**

Originada de solicitações de mudanças no ambiente, inclusão de novos requisitos, correções de defeitos e prevenção de problemas futuros, a manutenção consome boa parte do recurso disponível durante o ciclo de vida do software (BENNETT; RAJLICH, 2000). Com o passar do tempo, as constantes mudanças no software decorrentes de manutenções contribuem com a sua degeneração, provocando a redução da qualidade e dificultando cada vez mais a sua evolução (MENS *et al.*, 2010). Nesse contexto, a compreensão da evolução do software surge como elemento chave para apoiar o reestabelecimento da qualidade e facilitar manutenções futuras.

No entanto, a compreensão de como o software evoluiu pode ser dificultada quando são aplicadas reestruturações e refatorações[1] no software. Por exemplo, renomeação de variáveis, movimentação de blocos, adição ou remoção de parâmetros nas chamadas de métodos, dentre outras modificações, podem ser percebidas de forma isolada e não como parte de uma manutenção maior. Esse cenário pode tornar ainda mais complexa a compreensão da evolução do software ao longo do tempo.

Diante disso, algumas pesquisas se concentram na compreensão da evolução do software. Dentre essas pesquisas, há aquelas que visam caracterizar e simular o fenômeno da evolução de forma mais ampla (LEHMAN, 1980; ARAÚJO; MONTEIRO; TRAVASSOS, 2012). Outras visam fornecer ferramental para a compreensão das diversas pequenas mudanças que, quando combinadas, promovem a evolução de um software específico.

Essa segunda linha de pesquisas é o foco deste trabalho. Ela inclui abordagens de detecção de diferenças em artefatos textuais (HUNT; MCILROY, 1976; HORWITZ, 1990; APIWATTANAPONG; ORSO; HARROLD, 2004), que são capazes de representar as diferenças encontradas através de uma lista de alterações mínimas que transformam uma versão do artefato na outra. Apesar de genéricas em termos de linguagens de programação, essas abordagens são pouco precisas na identificação de modificações por não tratarem adequadamente modificações que afetam múltiplos artefatos.

Por outro lado, ainda na segunda linha de pesquisa citada anteriormente, há também abordagens de detecção de refatoração (DEMEYER; DUCASSE; NIERSTRASZ, 2000; DIG *et al.*, 2006; WEISSGERBER; DIEHL, 2006), que são capazes de identificar as refatorações

[1] As técnicas de refatoração (FOWLER *et al.*, 1999) são capazes de aprimorar atributos de qualidade do software sem alterar o seu comportamento.

realizadas entre duas versões. Essas abordagens ficam usualmente presas a um catálogo predefinido de refatorações e não consideram outros tipos de manutenção.

Este trabalho tem como principal motivação possibilitar a compreensão precisa de mudanças em artefatos textuais, mesmo quando essas mudanças envolveram reestruturações e refatorações do software. Essa motivação é de especial relevância por envolver um cenáriono qual as ferramentas de diff convencionais não têm um bom desempenho.

1.2 **OBJETIVO**

Diante do exposto, o objetivo deste trabalho é apresentar uma nova abordagem para detecção de diferenças em artefatos textuais, que seja independente de linguagem de programação, mas que forneça maior precisão na detecção de modificações decorrentes de refatorações. Para alcançar este objetivo, a abordagem proposta neste trabalho, denominada IDIFF, do inglês *Iterative diff,* foi norteada pelos seguintes propósitos:

Detecção da similaridade máxima global baseada em conteúdo: adoção de algoritmos de otimização, capazes de encontrar a correspondência entre artefatos que leva à similaridade global máxima entre as versões analisadas, considerando o conteúdo dos artefatos, em detrimento de seus nomes. Desta forma, é possível identificar todos os artefatos que possuem algum grau de similaridade com determinado artefato. Com isso, mudanças decorrentes de refatorações que transcendem a fronteira de um artefato ou que alteram o nome ou localização de um artefato podem ser identificadas.

Identificação precisa de mudanças: identificação de diferenças em granularidades distintas, possibilitando a detecção de movimentos de blocos compostos por arquivos (e.g., movimentação de classes para outros pacotes), linhas (e.g., movimentação de método entre classes) e palavras (e.g., movimentação de parâmetros de um método para outro).

Identificação eficiente de mudanças: refinamento automático de granularidades, partindo do grão mais grosso (arquivo) para o mais fino (caractere), com o objetivo de excluir grandes blocos semelhantes da comparação. Segundo Estublier (2000) o percentual de semelhanças entre versões consecutivas é próximo de 98%. Com o refinamento automático de granularidades é possível evitar a necessidade de um alto poder computacional quando

artefatos de tamanhos consideráveis são analisados, principalmente com a utilização de grãos finos (i.e., palavra e caractere).

Visualizações especializadas: adoção de visualizações com diferentes focos e perspectivas. Em relação ao foco, há a possibilidade de estabelecer um foco geral, no nível de diretórios, ou um foco detalhado, no nível de arquivos. O foco detalhado pode ser subdividido em comparação pareada (i.e., dois arquivos) e comparação múltipla (i.e., um arquivo com todos os demais). Em relação a perspectivas, dependendo do grau de similaridade entre os artefatos, é possível realçar a parte comum, útil para situações decorrentes de refatorações que transcendem a fronteira de um artefato, ou a parte diferente dos mesmos.

1.3 **QUESTÕES DE PESQUISA**

A construção da abordagem e a avaliação dos resultados através de experimentos têm como principal objetivo responder às seguintes questões de pesquisa:

- IDIFF aumenta a precisão quando comparado a ferramentas de *diff* convencionais?
- IDIFF aumenta a cobertura quando comparado a ferramentas de *diff* convencionais?
- Para quais tipos de refatoração o IDIFF possui resultados mais precisos?
- Qual é a parametrização de granularidade para IDIFF que apresenta resultados mais precisos?

1.4 **METODOLOGIA**

Inicialmente foi construído um mapeamento sistemático da literatura, com o objetivo de detectar as principais características presentes em ferramentas de diff considerando o estado da arte. O estudo permitiu a identificação dos algoritmos utilizados em abordagens desenvolvidas e principalmente a análise de falhas ou restrições na detecção de diferenças.

Em seguida, foi elaborado um survey com intuito de analisar o estado da prática, listando características essencias e desejáveis em ferramentas para detecção de diferenças. As ferramentas consideradas no survey foram selecionadas a partir de conhecimento prévio e utilização em linguagens de programação. A pesquisa foi realizada com pessoas do meio

acadêmico e do mercado de trabalho, permitindo uma análise de opiniões de forma ampla. Após análise do estado da arte e do estado da prática, o WinMerge foi selecionado como baseline para comparação com a ferramenta desenvolvida neste trabalho.

A abordagem IDIFF, proposta neste trabalho, foi baseada nas características encontradas considerando o estado da arte e o estado da prática. IDIFF visa a redução iterativa da granularidade de comparação, sem impor custos de processamento excessivos. De acordo com os artefatos de entrada, que são dois diretórios ou dois arquivos, IDIFF aciona o módulo correspondente, DDiff ou FDiff, respectivamente. O DDiff é responsável por avaliar o grau de similaridade global entre dois diretórios. Por outro lado, o FDIFF é responsável por identificar as diferenças no nível de arquivo, permitindo comparação pareada e múltipla. De forma simplificada, a abordagem é baseada em quatro etapas principais: filtragem, correspondência, comparação e visualização. Inicialmente, todos os arquivos idênticos e não comparáveis são filtrados. A partir disso, os demais arquivos são combinados de forma a alcançar a similaridade ótima global. Em seguida, cada par combinado dos arquivos é comparado de forma a identificar semelhanças considerando os grãos linha, palavra e caracter. Finalmente, os resultados podem ser visualizados em termos de semelhanças e diferenças de forma a facilitar a compreensão das mudanças realizadas.

Após a concepção da abordagem, uma ferramenta homônima foi construída. A ferramenta possibilita a parametrização de granularidades, separadores e limitadores. Na análise de diretórios, feita pelo módulo DDiff, o algoritmo de hash SHA1 é utilizado para atender a etapa de filtragem, identificando os arquivos idênticos. Além disso, os arquivos binários são selecionados como exclusões e inclusões de acordo com seu diretório. No que diz respeito a etapa de correspondência, o algoritmo Húngaro é utilizado com o objetivo de atingir a similaridade ótima global na comparação de arquivos pertencentes aos diretórios comparáveis. Para a comparação de arquivos, feita pelo módulo FDiff, o algoritmo LCS[2] é executado iterativamente, possibilitando a comparação pareada. A ferramenta disponibiliza telas específicas para cada comparação, possibilitando uma melhor visualização e compreensão das mudanças realizadas. Além da tela de comparação de diretórios, são disponibilizadas telas para comparação pareada, possibilitando a mudança de perspectivas para visualização dos resultados.

Após a construção da ferramenta, foram realizados experimentos com intuito de avaliar precisão, cobertura e aderência dos resultados encontrados, analisando principalmente

[2] *Longest Common Subsequence*

o comportamento da ferramenta diante dos cenários que apresentam resultados insatisfatórios em ferramentas convencionais. Os arquivos de entrada foram construídos a partir de exemplos expostos no livro de refatoração (FOWLER *et al.*, 1999), visando imparcialidade nos experimentos e enfatizando tipos de manutenção que fazem uso das técnicas de refatoração, uma vez que estas técnicas exploram justamente as características de difícil compreensão.

Após a execução dos experimentos, uma análise estatística foi realizada com utilização de variáveis dependentes capazes de calcular corretude e completude dos resultados, além do cálculo da aderência entre os resultados obtidos e os resultados esperados. Testes estatísticos para análises individuais e agrupadas mostram que os resultados obtidos pela ferramenta IDIFF são mais precisos quando comparados à ferramenta selecionada como baseline para o estudo.

1.5 **ORGANIZAÇÃO**

Este trabalho está organizado em cinco outros capítulos, além desta introdução. O Capítulo 2 expõe a base de conhecimento necessária para compreensão deste trabalho. Algoritmos de comparação são discutidos levando em consideração a análise de sequências (i.e., linhas) e conjuntos (i.e., arquivos) comuns. Em seguida, alguns aspectos sobre refatoração são descritos, com o intuito de permitir ao leitor uma melhor compreensão da análise dos experimentos realizados para avaliação deste trabalho. Finalmente são apresentados alguns trabalhos relacionados.

O Capítulo 3 apresenta a abordagem proposta, denominada IDIFF, iniciando por um exemplo motivacional propositalmente simples, para fins didáticos. A partir disso, os dois módulos que compõem a abordagem, DDiff e FDiff, são detalhados, visando a exposição dos algoritmos utilizados e do escopo de cada módulo na comparação de diferentes tipos de artefatos.

A ferramenta desenvolvida é apresentada no Capítulo 4. As subseções oferecem uma visão dos detalhes implementados, incluindo seleção dos artefatos a serem comparados e parametrização da ferramenta. Na sequência, as telas dos módulos DDiff e FDiff são exibidas, usando como base o projeto indicado no exemplo motivacional do Capítulo 3. A execução nos módulos DDiff e FDiff exemplifica os resultados obtidos pela ferramenta, de forma a explicitar os principais benefícios proporcionados pela utilização da mesma na análise de diferenças entre diretórios e arquivos. No módulo de análise de arquivos, FDiff, são relatadas as diferenças entre a comparação pareada e a comparação múltipla.

Após a descrição da abordagem e apresentação da ferramenta, o Capítulo 5 relata a avaliação realizada sobre o IDIFF. O planejamento e a execução dos experimentos são mencionados nas subseções, indicando como foi feita a obtenção e o tratamento dos dados para a avaliação. A análise dos resultados é detalhada através de testes de hipóteses. Por fim, são apontadas as ameaças a validade dos experimentos realizados.

Finalmente, o Capítulo 6 conclui o trabalho, enumerando contribuições, limitações e trabalhos futuros.

Capítulo 2 – **ALGORITMOS DE DIFF**

2.1 **INTRODUÇÃO**

A engenharia de software é uma área da computação que visa organização, produtividade e qualidade através de novas tecnologias e boas práticas (PRESSMAN, 2005). Segundo a engenharia de software, o processo de software passa por três fases essenciais (PRESSMAN, 2005): definição, desenvolvimento e manutenção. A fase de definição é responsável por identificar requisitos necessários, informações a serem processadas, expectativas, restrições e interfaces. Já a fase de desenvolvimento envolve, por exemplo, as estruturas, a definição de tecnologias a serem utilizadas, a arquitetura e bibliotecas de apoio. Por fim, a fase de manutenção contempla correções de defeitos, prevenção, adaptações e melhorias realizadas após implantação do software. Neste trabalho a terceira fase será abordada, principalmente no que diz respeito ao apoio à compreensão das manutenções realizadas durante o processo de software.

Os principais tipos de manutenção existentes no cenário de software podem ser descritos conforme a seguir (BENNETT; RAJLICH, 2000):

- **Manutenção corretiva:** Correção de defeitos do software, normalmente reportados pelos usuários do sistema.

- **Manutenção adaptativa:** Adaptação do software atendendo às solicitações de mudanças no ambiente, novos processadores, memórias, sistemas operacionais, permitindo o funcionamento e o aproveitamento de novos recursos. Além disso, é possível citar como exemplo as mudanças na legislação.

- **Manutenção perfectiva:** Aperfeiçoamento do software e de suas funcionalidades, proporcionando melhorias ao usuário.

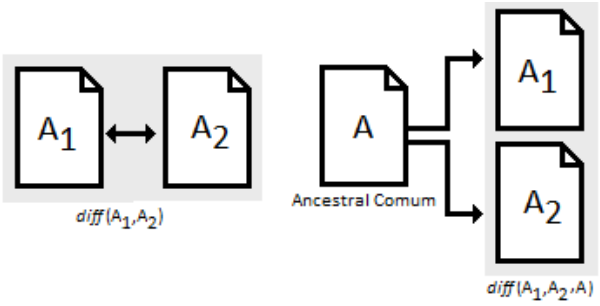
- **Manutenção preventiva (ou Reengenharia):** Redução da deterioração do software e simplificação das manutenções citadas anteriormente.

Diante do cenário da manutenção, é imprescindível abordar assuntos referentes à identificação das modificações para melhor compreensão de manutenções realizadas durante o processo de desenvolvimento, principalmente no que diz respeito ao trabalho colaborativo.

A seção 2.2 exibe algoritmos de comparação, introduzindo formas de detectar a maior sequência comum e otimizar alocação de recursos. As representações de diferenças são apresentadas na seção 2.3. Na seção 2.4 o tema refatoração é abordado, de forma a realçar a sua importância diante do cenário de manutenção de software. Por fim, a seção 2.5 descreve trabalhos relacionados e a seção 2.6 apresenta as considerações finais sobre os temas abordados neste capítulo.

2.2 **ALGORITMOS DE COMPARAÇÃO**

Os algoritmos de *diff* são utilizados para identificação de diferenças entre artefatos de software, conforme descrito por Mens (2002). De forma sucinta, o cálculo de diferenças pode fazer uso somente de duas versões sequenciais que estão sendo comparadas (*two-way diff*) ou adotar um ancestral comum para o caso onde as versões que estão sendo comparadas tenham sido criadas em paralelo (*three-way diff*). A Figura 1 exemplifica a diferença entre as duas classificações descritas. Para realização de *two-way diff* (A_1, A_2) apenas as duas versões A_1 e A_2 serão consideradas, conforme Figura 1(a). Já no *three-way diff* (A_1, A_2, A), apresentado na Figura 1(b), o ancestral comum A é utilizado para auxiliar na resolução de conflitos oriundos da detecção de diferenças entre as duas versões comparadas, criadas em paralelo.



(a) (b)

Figura 1 – Two-way diff e three-way diff

Desta forma, na análise *two-way diff* existe a possibilidade da ocorrência de conflitos ou mesmo ambiguidades para versões criadas em paralelo. Por exemplo, ações de adição, deleção e alteração de um trecho são difíceis de serem identificadas precisamente por

algoritmos *two-way diff*. Por outro lado, nem sempre há um ancestral comum disponível para comparação. Além disso, quando as versões são sequenciais (i.e., revisões), uma das versões é ancestral da outra, tornando *two-way diff* e *three-way diff* equivalentes. Como este trabalho trata da compreensão da evolução, onde as versões são essencialmente sequenciais, o escopo deste se encontra em *two-way diff*.

Porém vale ressaltar que identificação de diferenças passa pela detecção das similaridades entre os artefatos comparados. As próximas subseções descrevem algumas das principais características desses algoritmos de identificação de semelhanças entre artefatos sequenciais e não sequenciais.

2.2.1 **COMPARAÇÃO DE SEQUÊNCIAS**

Amplamente utilizado em aplicações biológicas, para comparação de DNA, o algoritmo de comparação de sequências tem como objetivo comparar duas sequências Y e Z e determinar suas semelhanças (CORMEN *et al.*, 2009). Seu funcionamento visa encontrar a sequência X de forma que seja constituída pela cadeia Y ou Z com zero ou mais elementos omitidos. Por exemplo, supondo as sequências: $Y = \{A, E, B, F, C\}$ e $Z = \{A, B, C, H\}$; $\{A\}$ e $\{A, B\}$ são exemplos de subsequências comuns das sequências descritas, enquanto {F} e {B, A} não são subsequências comuns das sequências descritas. Quando $\mathrm{X} = \max_{0 \le i \le n}\{X_i\}$, onde n é o número total de subsequências comuns existentes entre as duas cadeias, concluí-se que X é a maior subsequência comum de Y e Z.

As sequências, no caso de código fonte, quando algoritmos de *diff* textuais são utilizados, são representadas em termos de linhas, palavras ou caracteres de um arquivo. O efeito da utilização destas granularidades é o refinamento dos resultados obtidos. Normalmente com a utilização do grão fino (caractere) os resultados tendem a ser mais precisos, porém exige um maior tempo de processamento, o que pode se tornar inviável para projetos grandes.

O algoritmo LCS, do inglês Longest Common Subsequence, (CORMEN *et al.*, 2009) possibilita a identificação da maior sequência comum entre artefatos de software. A partir da maior sequência comum, é possível identificar as operações de adição e deleção que transformam a sequência comum em cada uma das sequências comparadas originalmente, resultando na diferença entre essas sequências.

Um possível algoritmo para encontrar a maior sequência comum consiste em obter todas as subsequências de um arquivo e verificar a existência no segundo arquivo. Esta

solução apresenta complexidade assintótica exponencial $O(2^n \times m)$ onde $O(2^n)$ é a complexidade requerida para gerar as subsequências distintas e $O(m)$ é a complexidade requerida para verificação da existência de uma subsequência no segundo arquivo (CORMEN *et al.*, 2009), com n e m representando os tamanhos dos arquivos em função da granularidade escolhida. Com foco na melhoria de desempenho, foi introduzido um algoritmo que faz uso de programação dinâmica para identificar a maior sequência comum entre duas sequências.

2.2.1.1 **PROGRAMAÇÃO DINÂMICA APLICADA AO LCS**

A programação dinâmica é um processo *bottom-up*, bastante usado em sistemas de otimização, para construção de soluções através do particionamento do problema em subproblemas, de forma que a solução ótima possa ser computada a partir das soluções previamente calculadas e memorizadas, evitando recálculo (CORMEN *et al.*, 2009).

A estrutura do algoritmo LCS, utilizado para comparar os arquivos x e y, é descrita na Equação 1 conforme apresenta por Cormen et. al. (2009).

Equação 1 – Algoritmo LCS com programação dinâmica

$$c[i,j] = \begin{cases} 0 & \text{se i } = 0 \text{ ou j } = 0 \\ c[i-1, j-1] + 1 & \text{se i , j } > 0 \; e \; \text{x}_\text{i} = \text{y}_\text{j} \\ \max(\, c[i, j-1], c[i-1, j]) & \text{se i , j } > 0 \; e \; \text{x}_\text{i} \neq \text{y}_\text{j} \end{cases}$$

Desta forma, c representa uma matriz $(n \times m)$, tal que n e m representam o tamanho de cada arquivo, em função da granularidade escolhida. Além disso, $c[i,j]$ armazena o tamanho da subsequência comum encontrada entre x_i e y_j, sendo x_i e y_j as subsequências de x e y da posição 1 até a posição i e j, respectivamente.

O funcionamento do algoritmo pode ser descrito através de um *loop* aninhado que incrementa i e j, de zero a n ou m, respectivamente, calculando o valor da célula $c[i,j]$ da matriz. A partir desta matriz, torna-se possível analisar todas LCS encontradas através da análise das subsequências analisadas. Este algoritmo gera complexidade assintótica reduzida, de $O(n \times m)$, alcançada através da aplicação de programação dinâmica. Desta forma o algoritmo possui complexidade quadrática, $O(n^2)$, supondo $n > m$.

Na execução do algoritmo, a célula pode ser composta por um número e uma seta, representando o valor de $c[i,j]$ e o caminho para a identificação dos elementos pertencentes à

LCS, respectivamente. A seta na diagonal é utilizada quando x_i e y_j são iguais. Caso contrário, a seta é direcionada para max $(c[i, j-1], c[i-1, j])$.

A Tabela 1 representa a execução com as sequências $Y = \{A, E, B, F, C\}$ e $Z = \{A, B, C, H\}$, onde cada célula representa o cálculo da maior sequência encontrada. O exemplo exposto é propositalmente simples para facilitar a compreensão do algoritmo.

Tabela 1 – Aplicação da programação dinâmica ao LCS

	Ø	**A**	E	**B**	F	**C**
Ø	0	0	0	0	0	0
A	0	↖ **1**	←1	←1	←1	←1
B	0	↑1	←1	↖ **2**	←2	←2
C	0	↑1	←1	↑2	←2	↖3
H	0	↑1	←1	↑2	←2	↑ **3**

Após conclusão da montagem da tabela, é possível percorrê-la através das setas. O processo de percurso para encontrar a LCS após a matriz estar pronta inicia na última célula da tabela, preenchida com o valor 3 no exemplo apresentado, o que indica que a maior sequência comum encontrada é de tamanho igual a 3. A partir disso, é possível identificar a maior sequência comum encontrada para este exemplo (i.e., $\{A, B, C\}$). Começando pela última célula, e seguindo as indicações das setas que compõem a tabela, é possível percorrer o caminho que indica os elementos que compõem a maior sequência comum. No exemplo apresentado, o caminho a ser percorrido está realçado, assim como os componentes da maior sequência comum.

O algoritmo LCS é capaz de identificar similaridades entre duas sequências. Contudo, quando os dados de entrada não são sequenciais, como os arquivos presentes em um diretório, este algoritmo não é eficaz.

2.2.2 **COMPARAÇÃO DE CONJUNTOS**

O contexto de comparação de conjuntos abordado neste trabalho diz respeito aos dois conjuntos de arquivos pertencentes aos diretórios que serão comparados. Cada arquivo é composto por um *path* e nome. É importante notar que a comparação de dois diretórios não pode ser considerada como um problema sequencial, uma vez que a ordenação dos arquivos

em um diretório, por exemplo, pode divergir em duas versões apresentadas, e o nome dos arquivos pode ser alterado entre duas versões. O algoritmo Húngaro, proposto por Kuhn (1955), foi utilizado como alternativa para tratamento de problemas não sequenciais neste trabalho.

O algoritmo Húngaro é um algoritmo de otimização combinatória para resolução do problema de alocação de tarefas, com complexidade assintótica $O(n^3)$. O principal objetivo é minimizar valores em uma matriz de custos, realizando assim a alocação ótima (KUHN, 1955).

Para execução do algoritmo, é necessária a construção de uma matriz c de dimensão $n \times n$, de tal forma que $c_{i,j}$ representa o custo de alocação da agente i na tarefa j. Para melhor compreensão, é apresentado um exemplo simplificado de execução do algoritmo e o resultado obtido a cada passo. A Tabela 2 apresenta matriz c contendo custos homem/hora para realização de três diferentes tarefas para quatro analistas de sistemas, sendo Analistas = {Ana, Maria, José, Pedro} e Tarefa = {Manutenção corretiva, Manutenção adaptativa, Manutenção perfectiva}.

Tabela 2 – Matriz de custos (Analistas x Tarefas)

	Ana	Maria	José	Pedro
Manutenção corretiva	20	30	40	70
Manutenção adaptativa	10	30	50	80
Manutenção perfectiva	5	20	10	90

1º Passo – Criação da Matriz: A matriz obrigatoriamente deve possuir o número de linhas igual ao número de colunas. Caso isso não seja verdade, torna-se necessária a adição de linhas ou colunas com custos nulos, conforme mostra a Tabela 3, onde a linha 4 foi incluída com esse objetivo. Os rótulos serão omitidos para melhor visualização da execução do algoritmo.

2º Passo – Redução de mínimo em linhas: Para cada linha i, encontre o vetor de custo x_i de tal forma que $x_i \leftarrow \min_{1 \leq i \leq n}\{c_{i,1} \dots c_{i,n}\}$. Em seguida, subtraia x_i em todos os valores contidos na linha i gerando valores relativos $\{c_{i,1} - x_i \dots c_{i,n} - x_i\}$, tal que $1 \leq i \leq n$. A Tabela 4 apresenta o resultado desse passo do algoritmo, considerando x_i = {20, 10,5,0}, calculado a partir da Tabela 3.

Tabela 3 – Criação da matriz de custos

20	30	40	70
10	30	50	80
5	20	10	90
0	0	0	0

Tabela 4 – Redução de mínimo em linhas

0	10	20	50
0	20	40	70
0	15	5	85
0	0	0	0

3º Passo – Redução de mínimo em colunas: Para coluna j, encontre o custo y_j de tal forma que $y_j \leftarrow \min_{1 \leq j \leq n}\{c_{1,j} \dots c_{n,j}\}$, em seguida subtraia y_j em todos os valores contidos na coluna j, gerando valores relativos $\{c_{1,j} - y_j \dots c_{n,j} - y_j\}$, tal que $1 \leq j \leq n$. Para todas as colunas desta matriz o valor mínimo é 0 (zero), com isso os valores permanecem inalterados conforme pode ser visto na Tabela 5.

Tabela 5 – Redução de mínimo em colunas

0	10	20	50
0	20	40	70
0	15	5	85
0	0	0	0

4º Passo – Seleção de entradas zero: Risque com o número mínimo de traços possíveis, todas as linhas e colunas que possuem custo zero. Em algumas situações mais de uma combinação de traços pode ser construída, porém o importante é que seja utilizado sempre o menor número de traços. A Tabela 6 exibe execução deste passo no exemplo proposto.

Após o 4º passo, torna-se necessário avaliar se o resultado apresentado é ótimo. Desta forma, se o número de traços utilizados para cobrir os custos iguais a zero for igual ao número de linhas da tabela, então a combinação ótima foi encontrada e o 7º passo deve ser executado.

Senão a execução dos passos na sequência deve ser realizada. No exemplo o número de traços ($n_{traços} = 2$) é inferior ao número de linhas ($n = 4$) da matriz.

Tabela 6 – Seleção de entradas zero

0	10	20	50
0	20	40	70
0	15	5	85
0	0	0	0

5º Passo – Adição de elemento mínimo: Adicione o elemento mínimo não marcado a cada elemento marcado. No exemplo proposto, temos como elementos marcados $\{c_{1,1} = 0,\ c_{2,1} = 0,\ c_{3,1} = 0,\ c_{4,1} = 0,\ c_{4,2} = 0,\ c_{4,3} = 0,\ c_{4,4} = 0\}$ e como elementos não marcados $\{c_{1,2} = 10,\ c_{1,3} = 20,\ c_{1,4} = 50,\ c_{2,2} = 20,\ c_{2,3} = 40,\ c_{2,4} = 70,\ c_{3,2} = 15,\ c_{3,3} = 5,\ c_{3,4} = 85\}$, tal que $c_{3,3} = 5$ é o mínimo entre os elementos não marcados.

É importante observar que se um elemento é marcado por m linhas, o valor mínimo deve ser somado m vezes, como acontece no elemento $c_{4,1}$ que será acrescido do valor 10 $(2 \times c_{3,3})$. Com a execução do 5º passo, temos uma matriz com novos valores, representados na Tabela 7.

Tabela 7 – Adição de valor mínimo não marcado

5	10	20	50
5	20	40	70
5	15	**5**	85
10	**5**	**5**	**5**

6º Passo – Subtrair elemento mínimo: Nesta etapa o menor valor existente em toda a tabela será usado para reduzir todos os demais valores, ou seja, $\forall c_{i,j} \in T, c_{i,j} = c_{i,j} - \min_{\substack{0 \le i \le n \\ 0 \le j \le n}} c_{i,j}$. No exemplo $\min_{\substack{0 \le i \le n \\ 0 \le j \le n}} c_{i,j} = 5$. A Tabela 8 apresenta elementos já reduzidos e uma nova análise de otimalidade.

Tabela 8 – Subtração de valor mínimo e nova análise de otimalidade

0	5	15	45
0	15	35	65
0	10	0	80
5	0	0	0

No exemplo após a execução dos passos citados, o número de traços ainda é inferior ao número de linhas da matriz, por esse motivo o 4º passo deve ser reexecutado assim como os passos consecutivos. A Tabela 9 resume o resultado após reexecução dos passos de acordo com o exemplo desenvolvido.

Tabela 9 – Reexecução dos passos 5 e 6

5	5	15	45
5	15	35	65
5	10	**5**	80
15	5	10	5

0	0	10	40
0	10	30	60
0	5	0	75
10	0	5	0

7º Passo – Seleção de solução ótima: selecionar células com valores zerados, de modo que não exista mais de um elemento marcado na linha e na coluna. Finalmente basta verificar resultados com os valores da tabela original, para encontrar o custo mínimo global. No exemplo proposto, os valores que compõem o custo mínimo são $\{c_{2,1} = 10,\ c_{1,2} = 30,\ c_{3,3} = 10\}$. A Tabela 10 representa a execução do 7º passo no exemplo proposto.

Tabela 10 – Análise e execução dos passos finais

0	**0**	15	40
0	10	35	60
0	5	**0**	75
10	0	5	0

	Ana	Maria	José	Pedro
Manutenção corretiva	20	30	40	70
Manutenção adaptativa	10	30	50	80
Manutenção perfectiva	5	20	10	90

Como pode ser visto, o exemplo exposto calculou o custo mínimo global, porém o algoritmo é facilmente adaptado para cálculo do máximo global. Para encontrar o máximo global, torna-se necessária a identificação, logo após o 1º passo, do maior elemento positivo da matriz tal que $\forall c_{i,j} \in T,\ e\_max = \max_{\substack{0 \le i \le n \\ 0 \le j \le n}} c_{i,j}$ gerando a partir dai uma matriz com novos custos, onde $c_{i,j} = (e\_max - c_{i,j})$. Os demais passos são idênticos aos citados anteriormente.

2.3 **REPRESENTAÇÃO DE DIFERENÇAS**

No cenário de software, os algoritmos de *diff* são utilizados principalmente no contexto de sistemas de controle de versão para redução do espaço de armazenamento e comparação de versões do software. O resultado produzido pelo *diff* entre duas versões do software é denominado delta. O restante dessa seção classifica algoritmos de deltas sob diferentes dimensões.

2.3.1 **DELTA SIMÉTRICO E DIRECIONADO**

Em relação a direcionalidade, delta simétrico pode ser descrito como o conjunto de diferença entre duas versões de tal forma que $\Delta\,(v_1, v_2) = (v_1 \backslash v_2)\ \cup\ (v_2 \backslash v_1)$, onde v_1 e v_2 são duas versões de um artefato de software e o operador \ indica subtração de conjuntos. Em contrapartida, delta direcionado refere-se a detecção de uma sequência de operações capaz de gerar uma versão a partir de outra através da aplicação do delta, ou seja, $v_2 = v_1 + \Delta\,(v_1, v_2)$.

2.3.2 **DELTA FÍSICO, SINTÁTICO E SEMÂNTICO**

Também é possível classificar deltas a partir das representações dos artefatos de software utilizados, como deltas textuais (físicos) (HUNT; MCILROY, 1976), sintáticos (YANG, 1991) e semânticos (JACKSON; LADD, 1994). O delta textual, gerado, por exemplo, pelo programa *Unix diff* [3], envolve duas versões de arquivos de texto e utiliza maior sequência comum para detecção de diferenças (algoritmo detalhado na seção 2.2.1). Apenas o delta textual pertence ao escopo deste trabalho. Quando representações sintáticas são

[3] http://www.gnu.org/software/diffutils/

processadas pelo *diff*, o delta é classificado como sintático. Por exemplo, Yang (1991) apresenta uma abordagem para comparação que explora o conhecimento da gramática da linguagem de programação, em que o delta é gerado em termos dos elementos pertencentes a essa linguagem. Finalmente, o *diff* semântico (JACKSON; LADD, 1994), que produz deltas semânticos, se propõe a compreender o objetivo da mudança através da análise da diferença semântica, expressa em termos do comportamento da entrada e saída.

2.3.3 **USO DE DELTA NO ARMAZENAMENTO DE VERSÕES**

Quanto ao uso de deltas por sistemas de controle de versão, no que tange armazenamento, os principais mecanismos são: *Complete*, *Forward*, *Backward* e *In-line* delta. A Figura 2 apresenta a representação dos diferentes tipos de armazenamento de versões.

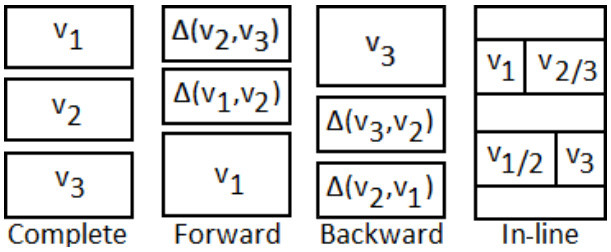


Figura 2 – Mecanismos de armazenamento de diff

O mecanismo *Complete delta* armazena versões de forma completa sem a utilização de deltas, apesar de recuperar versões de forma eficiente, necessita de um grande espaço para armazenamento das versões. No mecanismo de *Forward delta* a versão original é armazenada de forma completa, e, a partir dela, apenas os deltas são armazenados. Sendo assim, para obter a versão v_3, torna-se necessária a aplicação dos deltas sobre a versão original, ou seja: $v_3 = v_1 + \Delta(v_1, v_2) + \Delta(v_2, v_3)$. Já o *Backward delta* armazena a versão mais recente de forma completa; com isso a aplicação do delta só se torna necessária quando existe a necessidade de recuperar versões anteriores. Apesar do *Forward delta*, usado no SCCS (ROCHKIND, 1975), ser mais intuitivo, o *Backward* delta, utilizado RCS (TICHY, 1985), é mais eficiente na recuperação da versão mais atual. Finalmente *In-line delta* apresenta um único resultado, com marcações de adições e deleções, sem repetição da parte comum. Como exemplo de uso é possível citar o sistema de controle de versão ClearCase (WHITE, 2000).

2.4 **REFATORAÇÃO**

Refatorações visam melhorar a qualidade do software em termos de manutenibilidade e legibilidade. (FOWLER *et al.*, 1999). A refatoração se baseia em passos simples para aprimoramento da estrutura interna do código. Para exemplificar, a Figura 3 apresenta um método que realiza conversão de temperatura, mal estruturado e com baixa coesão, visto que realiza, além de duas conversões distintas, a impressão de valores convertidos.

```
public void convert(int celsiusTemperature, int fahrenheitTemperature) {
  int fahrenheit = celsiusTemperature * 9 / 5 + 32;
  int celsius = (fahrenheitTemperature - 32) * 5 / 9;
  System.out.println("Fahrenheit --> Celsius: " + celsius);
  System.out.println("Celsius --> Fahrenheit: " + fahrenheit);
}
```

Figura 3 – Método convert

A Figura 4 apresenta o resultado da refatoração, denominada *Extract Method*, aplicada ao método *convert.java*, segundo Fowler (1999) proporcionando maior coesão, reuso e legibilidade do código avaliado. É importante observar que a refatoração não altera o resultado esperado. No exemplo exposto, as duas versões, com e sem refatoração, apresentam a saída descrita na Figura 5 quando recebem o valor 300 para conversão da temperatura em ambas as escalas.

```
public void convert(int celsiusTemperature, int fahrenheitTemperature) {
  int fahrenheit = celsiusToFahrenheit(celsiusTemperature);
  int celsius = fahrenheitToCelsius(fahrenheitTemperature);
  printTemperature(celsius, fahrenheit);
}
private int fahrenheitToCelsius(int fahrenheitTemperature) {
  return (fahrenheitTemperature - 32) * 5 / 9;
}
private int celsiusToFahrenheit(int celsiusTemperature) {
  return celsiusTemperature * 9 / 5 + 32;
}
private void printTemperature(int celsius, int fahrenheit) {
  System.out.println("Fahrenheit --> Celsius: " + celsius);
  System.out.println("Celsius --> Fahrenheit: " + fahrenheit);
}
```

Figura 4 – Extract method aplicado ao método converter.java

Para garantir a corretude dos resultados é essencial a utilização de testes, que devem ser executados antes da alteração, para armazenamento de resultados esperados, e após a refatoração, garantindo que não ocorreram mudanças funcionais. Seguindo essa filosofia, a IDE Eclipse possui um plugin denominado *SafeRefactor* (SOARES, 2010) capaz de verificar se a refatoração a ser realizada é segura. Para isso, são gerados testes a fim de encontrar as mudanças de comportamento na aplicação da refatoração.

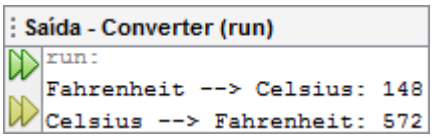


Figura 5 – Resultado da execução do método convert.java

Fowler (1999) apresenta grupos de acordo com o propósito de cada refatoração, todos descritos no Apêndice C.

Essas refatorações e seus grupos foram mantidos com nome em inglês para evitar ambiguidades decorrentes da tradução. No contexto deste trabalho, elas serão utilizadas durante a avaliação da abordagem proposta, como detalhado no Capítulo 5.

2.5 **TRABALHOS RELACIONADOS**

O mapeamento sistemático da literatura (PETERSEN *et al.*, 2008), foi realizado antes da elaboração da proposta, visando identificar características das abordagens no estado da arte. O planejamento e a execução deste mapeamento são apresentados no Apêndice A deste trabalho.

A partir da seleção dos artigos, dois grupos de trabalhos relacionados foram observados: o grupo que realiza detecção de diferenças e o que detecta movimentações entre artefatos. As próximas seções apresentam trabalhos relacionados de acordo com o agrupamento proposto.

2.5.1 **DETECÇÕES DE DIFERENÇAS**

As diferentes implementações do programa *diff* comparam artefatos, encontram diferenças, relatam os resultados e, em alguns casos, fornecem maneiras de suprimir

diferenças irrelevantes, como por exemplo, linhas em branco e comentários (HUNT; MCILROY, 1976; HUNT; SZYMANSKI, 1977; MILLER; MYERS, 1985; OBST, 1987).

Hunt et. al. (1976) apresentam uma implementação do programa *diff* que relata as diferenças entre dois arquivos, todas expressas como uma lista mínima de alterações, capazes de transformar um arquivo origem em um arquivo destino, considerando a linha como unidade indivisível.

Este algoritmo foi projetado para realização de uso eficiente de tempo e espaço, motivo pelo qual ainda é vastamente utilizado. O LCS é o algoritmo central desta proposta. Com a identificação da maior sequência comum, é possível encontrar linhas similares entre os arquivos comparados, e, posteriormente, identificar as diferenças.

Diversas técnicas foram utilizadas para obtenção de um melhor desempenho na detecção das similaridades, como, por exemplo, *hashing*, preordenação, programação dinâmica, dentre outras. A utilização de programação dinâmica é apresentada para detecção da maior sequência comum, devido a capacidade de comparar arquivos grandes.

Após aproximadamente um ano, Hunt et. al. (1977) apresentaram um algoritmo para detecção da maior sequência comum com uma melhor complexidade assintótica. Segundo estudo apresentado, o tempo de execução no pior caso é da ordem $O(n \times \log n)$ enquanto algoritmos anteriores apresentam a complexidade igual a $O(n^2)$, onde n representa o comprimento das duas sequências. Na proposta apresentada, a complexidade em relação ao tempo de execução é de $O\ ((r\ +\ n)\ log\ n)$, onde r é o número total de posições de pares ordenados nas quais as duas sequências coincidem. Logo, na análise de pior caso temos que $r\ = n$, com isso, o algoritmo tem um tempo de execução de $O\ (n\ \times \log n)$. A proposta representa as diferenças encontradas em termos de operações de inserção e exclusão.

Miller et at. (1985) apresentam um programa de comparação de arquivos baseado no cálculo da menor sequência de comandos de inserção e exclusão capazes de converter um determinado arquivo em outro. Segundo o trabalho apresentado, o método é particularmente eficiente se a diferença entre os arquivos é reduzida quando comparada ao tamanho dos arquivos analisados.

O programa produz uma lista de diferenças entre os dois arquivos comparados, sendo que as diferenças são expressas em termos de linhas, de forma a indicar quais linhas devem ser inseridas, apagadas ou movidas para converter o arquivo base no arquivo comparado, conforme propostas anteriores. Assim como os trabalhos apresentados anteriormente, o grão linha é considerado como unidade indivisível, de forma que um arquivo de n linhas seja analisado como contendo uma sequência de n símbolos. O algoritmo apresentado na proposta

produz o menor roteiro possível de edição. A análise demonstra que o algoritmo é mais eficiente quando comparado ao *Unix diff*, porém funciona bem com diferenças reduzidas entre arquivos, cenário mais comum na manutenção de software. Quando este algoritmo é aplicado a arquivos pouco similares, a sua eficiência fica comprometida.

2.5.2 **DETECÇÃO DE MOVIMENTAÇÃO**

Para contemplar cenários comuns durante o ciclo de vida do software, a detecção de movimentação de blocos (TICHY, 1984) torna-se cada vez mais necessária, assim como a detecção de refatorações (DIG *et al.*, 2006; WEISSGERBER; DIEHL, 2006) e de clones (MALPOHL; HUNT; TICHY, 2003).

Tichy (1984) apresenta o problema de correção *string* para *string* onde o objetivo é encontrar um mínimo de sequência de operações para transformar uma sequência na outra. Neste trabalho, é proposta a solução denominada *bdiff*, um aperfeiçoamento do algoritmo da maior sequência comum, LCS, através da aplicação iterativa do algoritmo.

A estratégia consiste em detectar a maior sub-sequência comum, em seguida retira-la da sequência analisada e computá-la novamente, utilizando a sequencia remanescente. Esses passos são repetidos até que não sejam encontradas subsequências comuns, ou seja, a subsequência comum seja de comprimento igual a zero. Desta forma, todas as possíveis sequências comuns são detectadas, possibilitando a identificação do menor conjunto de diferenças existentes.

Na análise, observou-se que a velocidade era similar ao *Unix diff*, porém produzindo deltas 7% menores. A conclusão da baixa redução do percentual é que movimentação de blocos e duplicidade de linhas não eram operações frequentes o suficiente para economia significativa de espaço. Porém com a utilização iterativa do algoritmo torna-se possível a detecção de blocos movidos entre dois artefatos comparáveis, o que é útil quando o objetivo é compreensão.

No cenário de detecção de refatorações, Malpohl et al. (2003) construíram uma ferramenta que detecta automaticamente atributos renomeados compondo parte de um conjunto de ferramentas de *diff* e *merge* que exploram a semântica estática de linguagens de programação. Inicialmente, a ferramenta analisa cada arquivo individualmente, associando cada declaração de atributos com todas as referências de uso. Em seguida, relaciona declarações nas duas versões comparadas utilizando informações de contexto, como, por exemplo, tipos de variáveis e semelhança de cadeias. Um fator importante desta proposta é a

independência de linguagem, quando consideramos o núcleo do detector de refatoração. A ferramenta, insensível à formatação, é passível de adaptação para reconhecimento de outras linguagens através de identificação de símbolos e algumas regras específicas de linguagem para análise da renomeação.

Dig et al. (2006) apresentam um algoritmo para detecção confiável de refatorações, que tem por finalidade a melhor compreensão do programa. O algoritmo utiliza a combinação de uma rápida análise sintática, para detecção de candidatos a refatoração, e uma análise semântica mais custosa, capaz de refinar os resultados obtidos na análise anterior. Para realização dessas verificações, torna-se necessário e imprescindível um registro de refatorações para a reprodução das mesmas.

Por se tratar de identificação exata da refatoração realizada, apenas sete tipos distintos de refatoração foram analisados. Os experimentos indicam resultados positivos em um cenário real, onde os artefatos são reutilizados com frequência através do trabalho colaborativo, e um número reduzido de falsos positivos, inferior a 10%.

Weissgerber et al. (2006) apresentam uma técnica para detecção de alterações que sejam possíveis refatorações, e as classificam de acordo com a probabilidade. Dois tipos de refatorações são utilizados: estruturais e locais. O protótipo construído detecta as seguintes refatorações estruturais: *Move Class, Move Interface, Move Field, Move Method, and Rename Class*. Além dessas, as seguintes refatorações locais também são detectadas: *Rename Method, Hide Method,Unhide Method, Add Parameter, and Remove Parameter*. A proposta realiza análise sintática e análise baseada em assinaturas para avaliação destas refatorações. Após a avaliação, é possível verificar o alto grau de identificação de refatorações. Os resultados mostram que em algumas categorias foi obtida cobertura de 77% e precisão de 73%. Considerando refatorações estruturais os valores passam a ser de 80% e 92% respectivamente.

2.6 **CONSIDERAÇÕES FINAIS**

Este capítulo apresentou algoritmos relevantes para identificação de similaridades e definição de uma melhor forma de alocação de recursos, compondo assim a base para a proposta deste trabalho. Em seguida discutiu algumas soluções existentes para o problema de detecção de diferenças e movimentações. Diante disso, dois principais pontos motivaram a construção de uma nova abordagem:

- A percepção das modificações que relacionam um arquivo com outros, ou seja, análise de similaridades entre arquivos, considerando conteúdo em detrimento do nome;

- Independência de linguagem de programação e refatorações específicas na detecção de similaridades entre artefatos de software.

A partir disso é proposta uma nova solução para detecção de diferenças entre artefatos sequencias e não sequenciais, independente de linguagens e refatorações, que possibilita análise entre arquivos. Esta abordagem, denominada IDIFF, é apresentada no Capítulo 3.

Capítulo 3 – **IDIFF**

3.1 **INTRODUÇÃO**

O Capítulo 2 identificou a necessidade de abordagens independentes de linguagem de programação que façam detecção de blocos movidos dentro ou entre arquivos considerando a similaridade dos mesmos. Diante disso, é apresentada uma abordagem destinada a apoiar a compreensão de como duas versões de software diferem, mesmo quando refatorações são realizadas. Essa abordagem visa priorizar a precisão dos resultados obtidos, sem a necessidade de alto poder computacional.

Este capítulo tem como principal objetivo apresentar a abordagem proposta para este trabalho. A seção 3.2 exibe um exemplo motivacional, intencionalmente simples, que proporciona uma visão dos problemas gerados na detecção de diferenças por conta do uso de refatorações. A seção 3.3 descreve a abordagem IDIFF juntamente com seus módulos, DDiff e FDiff, para tratamento de diretórios e arquivos, respectivamente. Por fim, a seção 3.4 exibe considerações finais referentes aos assuntos expostos.

3.2 **EXEMPLO MOTIVACIONAL**

Um ponto de venda, conhecido pela sigla PDV, refere-se a cada um dos caixas de um supermercado ou de uma loja. Com o avanço da tecnologia, surgem os softwares de PDV para transferência eletrônica de fundos, integrando o sistema do estabelecimento com o da rede compradora.

Todas as refatorações aplicadas neste cenário são de simples compreensão e foram utilizadas com o intuito de possibilitar a análise dos principais aspectos da abordagem proposta. A visualização das refatorações é representada através da escala de cores exibida na Tabela 11.

Tabela 11 – Tabela de cores

Exclusão	Adição	Movimentação	Modificação

O exemplo apresentado nesta seção, adaptado de Larman (2005), é baseado em duas versões de um sistema de PDV, implementado na linguagem Java seguindo princípios da programação orientada a objetos. A Figura 6 apresenta o diagrama de classes da versão base

do sistema. As principais entidades a serem analisadas estão demarcadas de acordo com as cores representadas anteriormente.

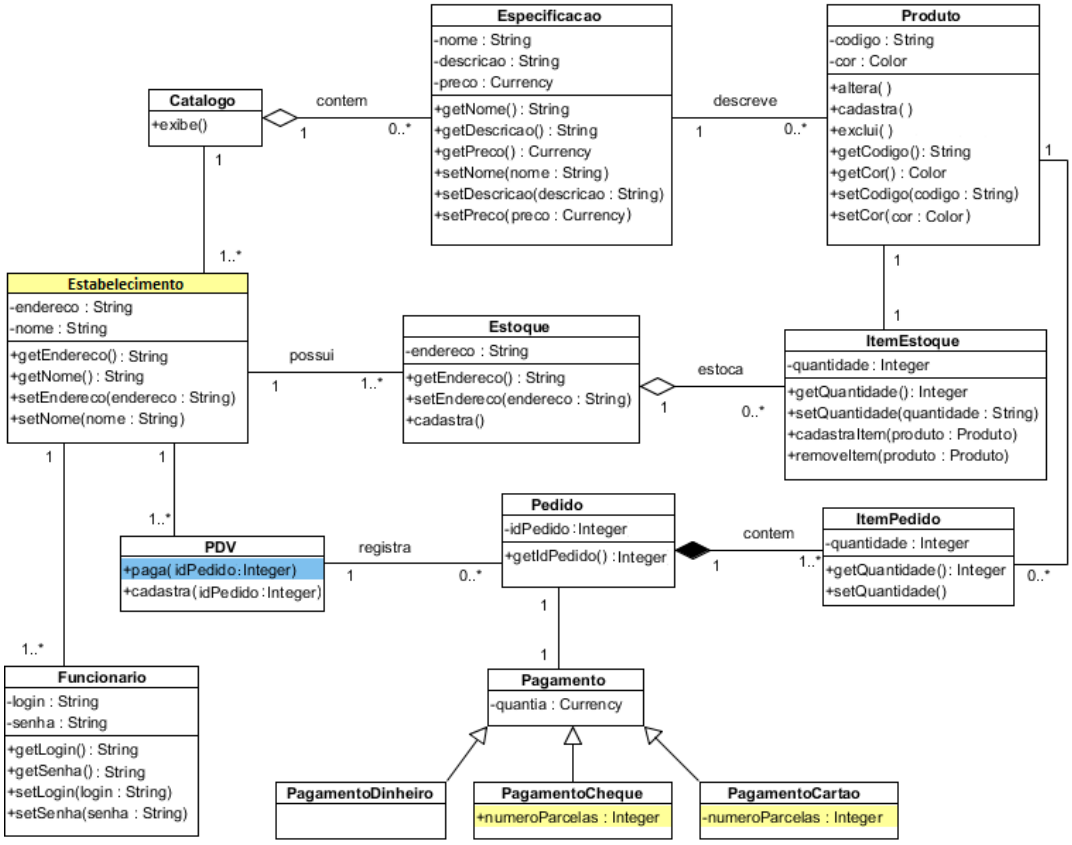


Figura 6 – Diagrama de classes da versão base do sistema PDV

Por sua vez, a Figura 10 apresenta o diagrama de classes do sistema após a aplicação de técnicas de refatoração, utilizadas para melhorar a estrutura interna do código. As refatorações aplicadas no exemplo exposto foram selecionadas por caracterizar a comparação de classes considerando conteúdo e por trabalhar em diferentes granularidades.

É importante observar que, apesar da utilização de diagramas para facilitar a compreensão, estes são artifícios meramente visuais, para representar os artefatos do código fonte em si. A partir da versão base, três diferentes tipos de refatorações foram aplicados:

Move Method: técnica de refatoração amplamente utilizada durante o desenvolvimento de software, que permite a redistribuição de responsabilidades entre as classes do projeto, aumentando a coesão e reduzindo o acoplamento do código implementado, segundo Fowler (1999). A Figura 7 apresenta a representação da refatoração aplicada entre duas classes no sistema PDV. No exemplo proposto o método *+paga(Integer idPedido)* foi

movido da classe *PDV.java* para a classe *Pedido.java.* Após movimentação do método o parâmetro *idPedido* torna-se desnecessário, porém, para manter a simplicidade do exemplo, ele foi mantido.

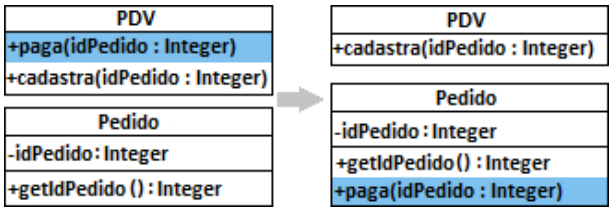


Figura 7 – Movimentação de método no programa PDV

Rename Method: técnica que permite a melhoria na compreensão de um código, de acordo com Fowler (1999). Amplamente utilizada para renomear métodos, neste exemplo foi aplicada para alterar nome de classe, desta forma, a classe *Estabelecimento.java* teve seu nome alterado para *Loja.java* conforme pode ser visto na Figura 8.

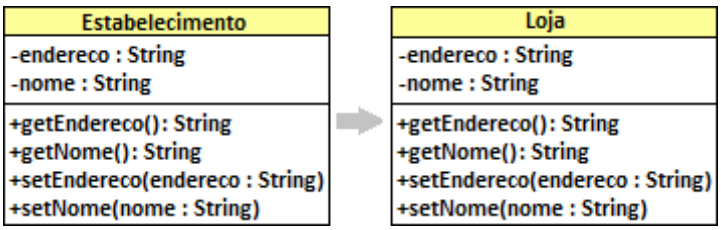


Figura 8 – Classe do sistema PDV renomeada

Encapsulate Field: segundo Fowler (1999), um dos mais relevantes princípios da programação orientada a objetos é o encapsulamento dos atributos de uma classe. Ou seja, os dados não devem ser usados de forma pública para manter a modularidade do programa. Para realização desta refatoração, inicialmente, o atributo passa de público para privado. Em seguida, são criados métodos de acesso para leitura e escrita do atributo. Por fim, todas as ocorrências de leitura e escrita, até então realizadas de forma direta ao atributo, são substituídas por chamadas aos métodos de acesso. A Figura 9 exibe a aplicação desta refatoração.

No exemplo apresentado, esta refatoração foi aplicada nas classes que realizam pagamento. Na classe *PagamentoCheque.java* o atributo *numeroParcelas* tornou-se privado, e os métodos *+getNumeroParcelas()* e *+setNumeroParcelas(numeroParcelas:String)* foram criados, possibilitando acesso ao atributo.

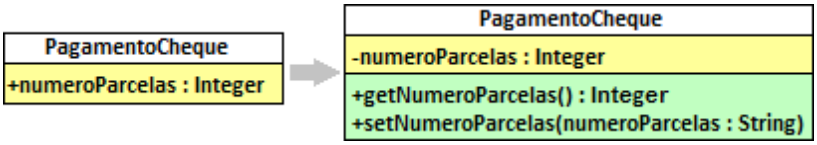


Figura 9 – Atributo encapsulado na classe PagamentoCheque.java

Em relação a classe *PagamentoCartao.java*, o atributo privado se tornou público permitindo acesso ao mesmo. Esta alteração simula uma alteração realizada sem aplicação de técnicas de refatoração e com utilização incorreta dos princípios de orientação a objeto. O resultado da aplicação das refatorações citadas anteriormente está representado no diagrama de classes da Figura 10.

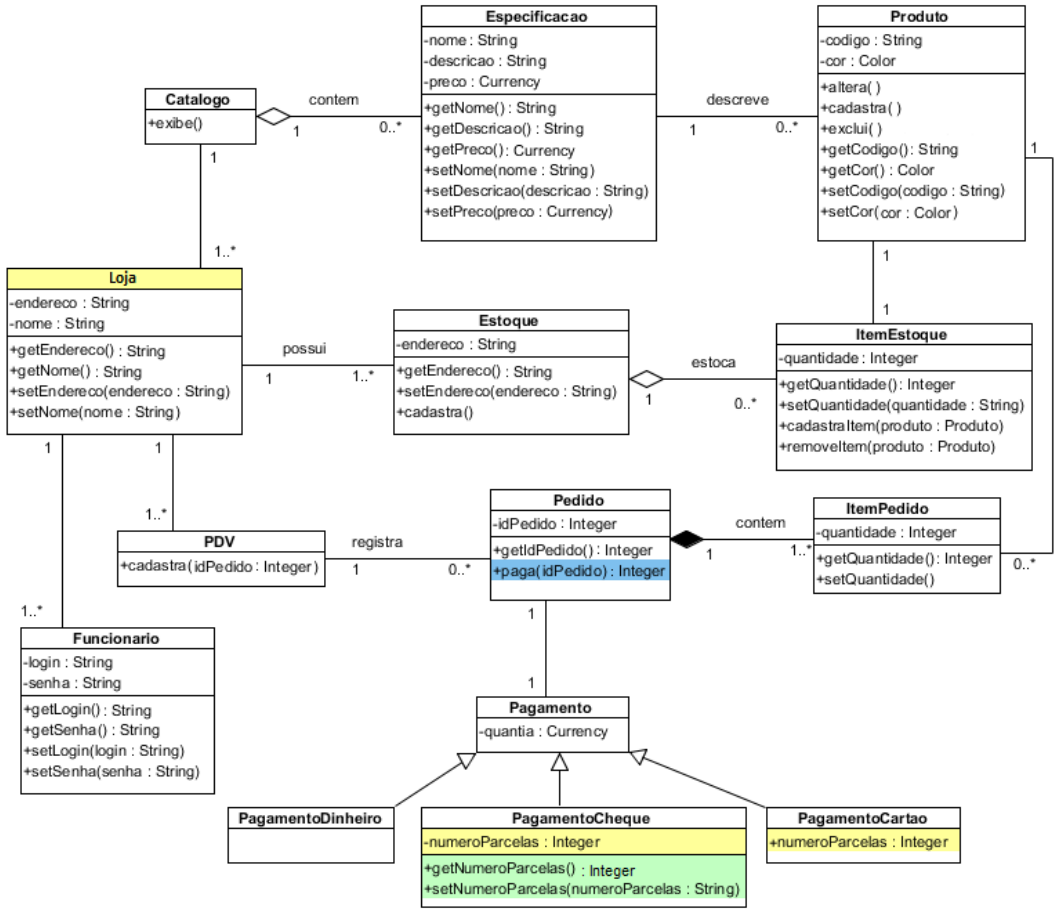


Figura 10 – Versão do sistema PDV após aplicação de técnicas de refatoração.

Ao analisar as duas diferentes versões do código com o uso de algoritmos convencionais de *diff*, as alterações normalmente são descritas em termos de adições e deleções. Além disso a maior parte destas ferramentas é restrita aos limites do arquivo, impossibilitando assim a identificação de movimentação de métodos entre classes. Como

exemplo de ferramenta tradicional é possível citar o programa *GNU diffutils*[4], composto pelos utilitários *diff*, *diff3*, *sdiff* e *cmp*.

A Figura 11 apresenta os resultados obtidos, em termos de adição e deleção, na comparação entre as classes envolvidas na movimentação do método *+paga(Integer idPedido)*. A ausência da detecção da movimentação fornece uma percepção incompleta da alteração realizada. Por sua vez, algumas ferramentas são capazes de detectar movimentações (TICHY, 1984), porém utilizam o arquivo, identificado pelo seu nome, como elemento chave para comparação. Em nosso exemplo, estas ferramentas não iriam apresentar resultados relevantes uma vez que a movimentação do método ultrapassou as fronteiras do arquivo.

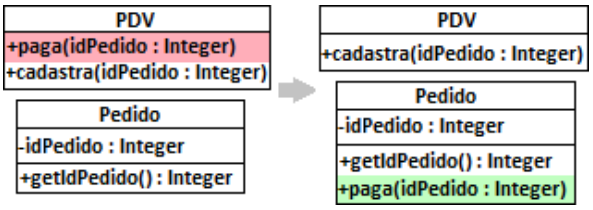


Figura 11 – Movimentação de método em ferramentas de diff.

O mesmo problema ocorre na identificação da alteração do nome de uma classe. A maioria das ferramentas apresenta como resultado exclusão da classe antiga e adição da nova classe, quando seria ideal detectar apenas a alteração do nome da classe. A Figura 12 apresenta de forma gráfica este cenário.

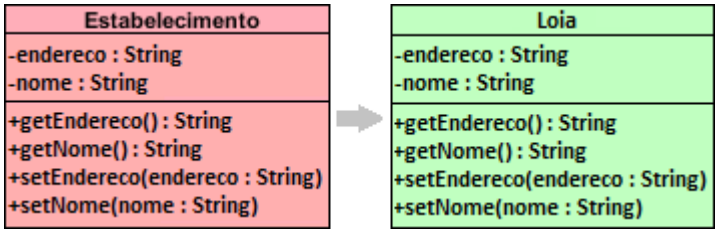


Figura 12 – Alteração do nome da classe não detectado por ferramentas tradicionais

Finalmente quando consideramos o cenário do encapsulamento, a maioria das ferramentas apresenta resultado satisfatório por se tratar de alteração no nível de arquivo, ou seja apenas uma comparação textual é suficiente para identificar a diferença.

[4] http://www.gnu.org/software/diffutils/

3.3 **IDIFF**

A abordagem denominada IDIFF, é proposta com o objetivo de apoiar a compreensão das mudanças ocorridas em diferentes versões de um artefato, sem a imposição de custos de processamentos excessivos. Para isso é realizada a redução iterativa de granularidades na unidade de comparação, partindo da granularidade mais grossa até a granularidade mais fina. A Figura 13 apresenta as granularidades utilizadas nesta abordagem e a ordem de refinamento.

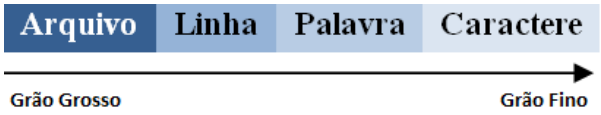


Figura 13 – Sentido do refinamento de granularidades

As granularidades expostas motivaram a criação de dois módulos distintos, DDiff e FDiff, conforme apresentado no diagrama de sequência da Figura 14. O módulo DDiff analisa grãos grossos (Diretório e Arquivo) e o módulo FDiff analisa grãos mais finos (Arquivo, Linha, Palavra e Caractere).

Figura 14 – Diagrama de atividades - IDIFF

O módulo DDiff é responsável por avaliar o grau de similaridade global entre diretórios. Por outro lado, FDiff é responsável por identificar diferenças no nível de arquivo, permitindo não só a comparação de um par de arquivos, mas também a comparação entre um arquivo e todos os outros arquivos existentes, quando acionado por meio do DDiff. A abordagem IDIFF é baseada em quatro etapas principais, iniciadas a partir da seleção de dois diretórios a serem comparados, conforme Figura 15.

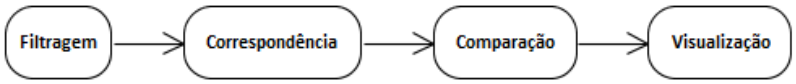


Figura 15 – Etapas da abordagem IDIFF

Filtragem: Todos os arquivos idênticos nos dois diretórios são filtrados, com o objetivo de reduzir o número de arquivos a serem comparados, evitando alto custo de comparação. Esta etapa é de responsabilidade do módulo DDiff e é realizada através da utilização de um algoritmo de *hash*. Além da identificação de arquivos idênticos também é realizada a filtragem dos arquivos não comparáveis, como, por exemplo, arquivos binários. É importante lembrar que a abordagem proposta neste trabalho é desenvolvida para artefatos textuais.

Correspondência: Todos os arquivos restantes são comparados de forma pareada tal que a semelhança global dos diretórios seja alcançada. Nesta etapa são realizadas comparações dois a dois, através de um produto cartesiano, para identificação do grau de similaridade entre todos os arquivos comparáveis e, consequentemente, dos diretórios que estão sendo comparados. A otimização global é detectada através da utilização do algoritmo Húngaro (KUHN, 1955) .

Comparação: Cada par combinado de arquivos é comparado de forma a identificar semelhanças relacionadas aos grãos linha, palavra e caractere. Esta etapa é de responsabilidade do módulo FDiff, que utiliza de forma iterativa, para cada grão analisado, o LCS com programação dinâmica para detecção de similaridades. A partir disso, o módulo identifica as diferenças entre artefatos comparados.

Visualização: Os resultados podem ser visualizados em termos de semelhanças e diferenças, com o intuito de facilitar a identificação das alterações realizadas durante a edição dos arquivos, possibilitando foco no que é relevante entre os arquivos comparados.

As etapas de filtragem e correspondência são apresentadas na subseção 3.3.1. A seção 3.3.2 apresenta a etapa de comparação e finalmente a seção 3.3.3 apresenta detalhes sobre a visualização dos resultados.

3.3.1 **DDIFF**

Inicialmente, é necessária a padronização da denominação das versões comparáveis de um artefato. Neste trabalho, a versão base, utilizada na comparação, é chamada de artefato à esquerda, enquanto a revisão desta versão é nomeada artefato à direita, a mesma nomenclatura é utilizada na comparação entre arquivos.

Retomando ao módulo DDiff, a Figura 16 apresenta o diagrama de atividades que representa todas as etapas do módulo DDiff, descritas de forma detalhada a seguir.

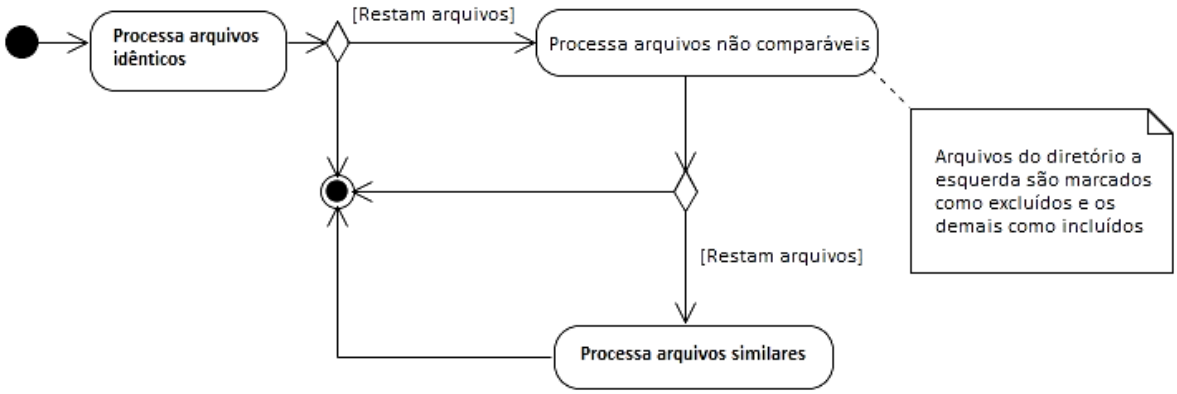


Figura 16 – Diagrama de atividades - DDiff

A primeira etapa, responsável pela detecção da similaridade global de diretórios, compreende a filtragem de arquivos idênticos. A Figura 17 ilustra em detalhe o processamento através de um diagrama de atividades.

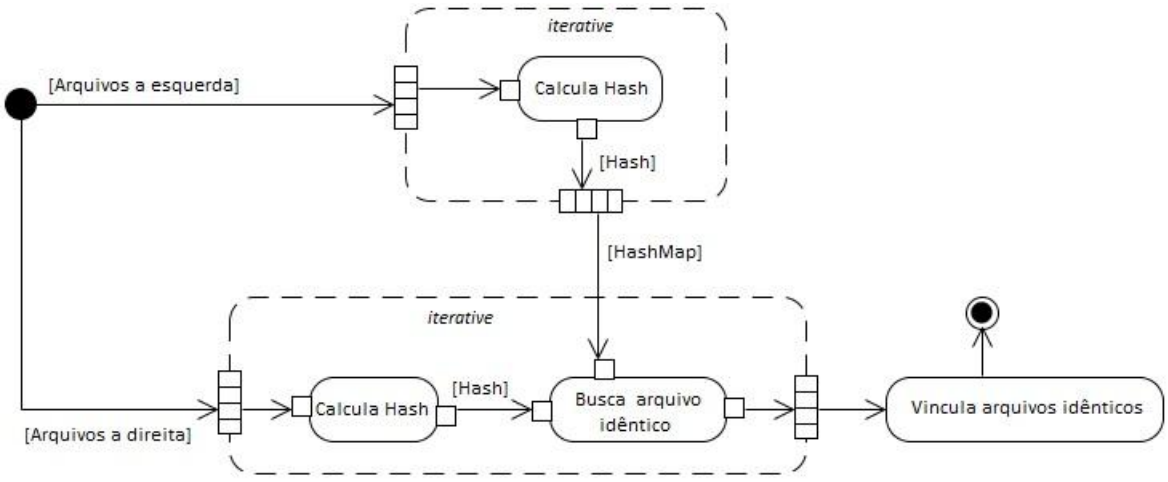


Figura 17 – Diagrama de atividades – Processa arquivos idênticos

Inicialmente um *hash* é calculado, de forma recursiva, para cada arquivo contido no diretório à esquerda, incluindo todos os seus subdiretórios, com um custo de $O(N)$. A Figura 18 apresenta, como exemplo, a comparação de dois diretórios e o cálculo dos hashs dos arquivos que compõem o diretório à esquerda.

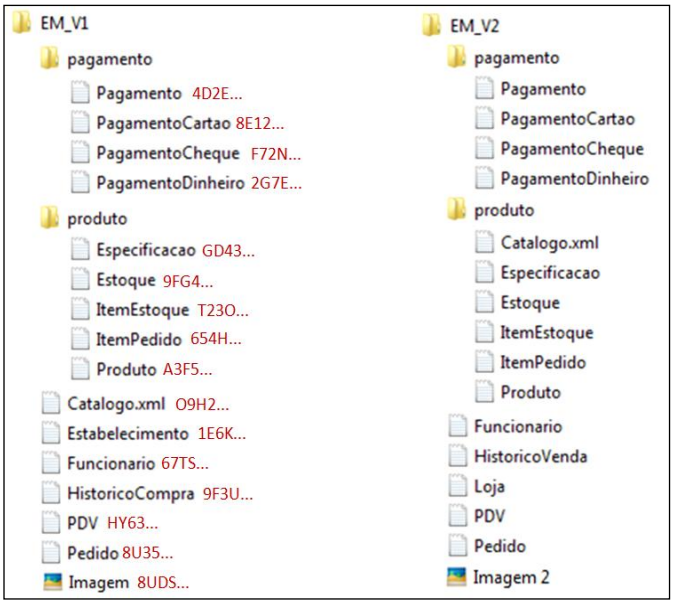


Figura 18 – Cálculo de hashs dos arquivos à esquerda

Em seguida, o mesmo processo é executado para cada arquivo à direita em $O(M)$. A partir disso, uma consulta é realizada para verificação de igualdade entre os conjuntos de *hash* calculados, com um tempo fixado em O (1). Neste contexto, N e M são, respectivamente, o número original de arquivos nos diretórios do lado esquerdo e do lado direito. Continuando o exemplo, a Figura 19 representa o cálculo do hash para cada arquivo que compõe o diretório à direita, e a identificação de arquivos idênticos entre os diretórios.

É importante frisar que todos os arquivos são considerados, independentemente da estrutura de subdiretórios. Ou seja, as listas de arquivos da esquerda e da direita contêm elementos com um determinado *path*, nome e conteúdo, porém nossa proposta é baseada em conteúdo, independentemente do *path* e nome dos arquivos. Após a execução deste passo, todos os arquivos idênticos foram filtrados em $O(N)$, conforme pode ser visto na Figura 20.

A partir disso, o módulo continua sua análise caso ainda possua arquivos a comparar. O próximo passo consiste na identificação de arquivos não comparáveis. Todos os arquivos binários são processados nesta etapa, sendo identificados como adições (se existirem apenas

no diretório à direita) ou deleções (se existirem apenas no diretório à esquerda). A Figura 21 destaca os arquivos que serão eliminados nesta etapa.

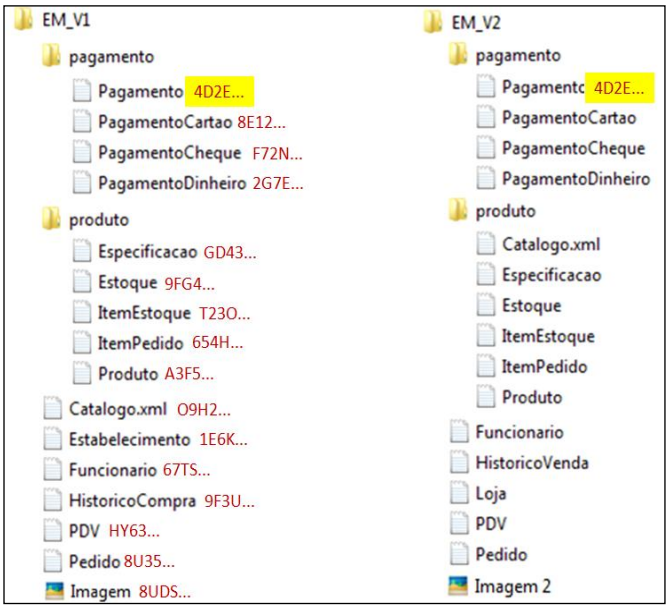


Figura 19 – Busca de arquivos idênticos

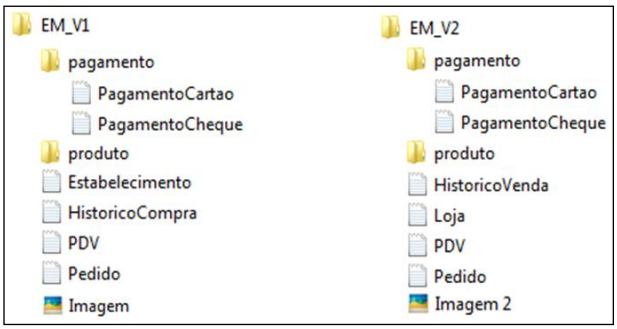


Figura 20 – Resultado da filtragem de arquivos idênticos

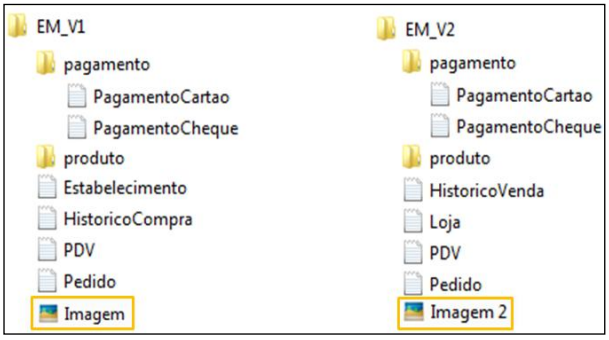


Figura 21 – Eliminação de arquivos não comparáveis

Novamente, caso ainda existam arquivos comparáveis, o processo continua. Após a execução dos passos anteriores, restam apenas os arquivos comparáveis com algum grau de similaridade, conforme pode ser visto na Figura 22

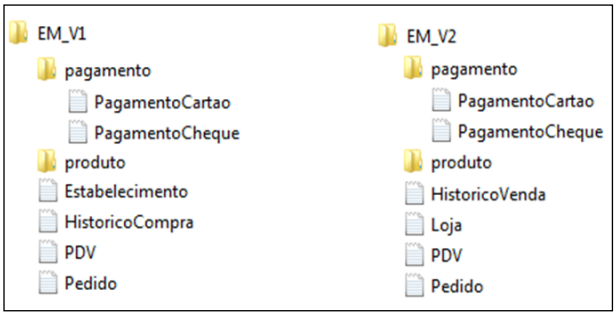


Figura 22 – Arquivos comparáveis selecionados pela etapa de filtragem

Em seguida, a comparação em pares é realizada, ou seja, cada arquivo à esquerda é comparado com todos os arquivos à direita com um custo $O(n \times m)$, onde n e m são respectivamente o número de arquivos restantes nos diretórios à esquerda e à direita após os passos de filtragem. Ou seja, suponha os dois conjuntos de arquivos restantes na esquerda e na direita, $\{E_1 \dots E_n\}$ e $\{D_1 \dots D_m\}$, compostos por n e m arquivos, respectivamente. A comparação conjunta é realizada de forma a identificar todas as semelhanças entre os arquivos comparáveis, de forma que $\forall_{1 \le i \le n} \forall_{1 \le j \le m} FDiff(E_i, D_j) = \frac{2 \times ILCS(E_i, D_j)}{size(E_i) + size(D_j)}$; Esta comparação, feita pelo módulo FDiff, produz uma matriz de dimensão $n \times m$, onde as linhas representam cada arquivo à esquerda, as colunas representam arquivos à direita, e as células representam a similaridade entre os arquivos, conforme pode ser visto na Figura 23 em um exemplo 3×3.

		Arquivos a direita		
		D_1	D_2	D_3
Arquivos a esquerda	E_1	FDiff(D_1,E_1)	FDiff(D_2,E_1)	FDiff(D_3,E_1)
	E_2	FDiff(D_1,E_2)	FDiff(D_2,E_2)	FDiff(D_3,E_2)
	E_3	FDiff(D_1,E_3)	FDiff(D_2,E_3)	FDiff(D_3,E_3)

Figura 23 – Matriz de comparação de arquivos

A Figura 24 representa o resultado do cálculo de similaridades entre a classe PagamentoCartao.java e todas as classes que compoem o diretório à direita.

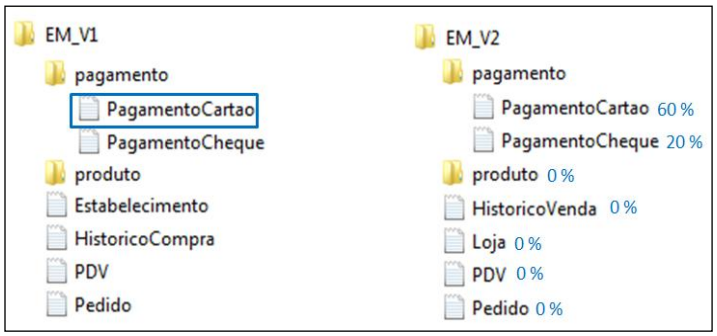


Figura 24 – Processamento de arquivos similares – comparação pareada

A matriz descrita é utilizada como entrada para o algoritmo Húngaro, que calcula uma combinação ótima entre os arquivos em $O(n^3)$, conforme pode ser visto no exemplo apresentado na Figura 25. Caso a matriz $n \times m$ não seja quadrada, a matriz é transformada para uma matriz quadrada $n \times n$, assumindo $n > m$, conforme apresentado no Capítulo 2. A Figura 26 descreve cada passo da etapa de identificação de arquivos similares.

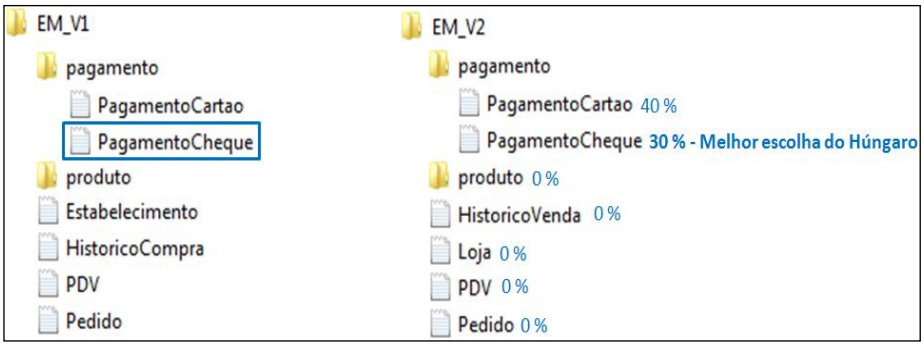


Figura 25 - Escolha do Húngaro para a classe PagamentoCheque.java

Neste ponto, é importante ressaltar que a complexidade do algoritmo, apesar de ser cúbica, não representa um problema para a abordagem, pois a etapa de filtragem evita o processamento arquivos idênticos e não comparáveis, reduzindo o número de arquivos comparados nesta etapa.

Finalmente, todos os arquivos com similaridade inferior a um limiar pré-definido são identificados em $O\ (n)$, sendo marcados como deleções e adições independentes. Este limiar pode variar de 0% a 100% e tem de ser parametrizado a fim de evitar a detecção de semelhanças irrelevantes.

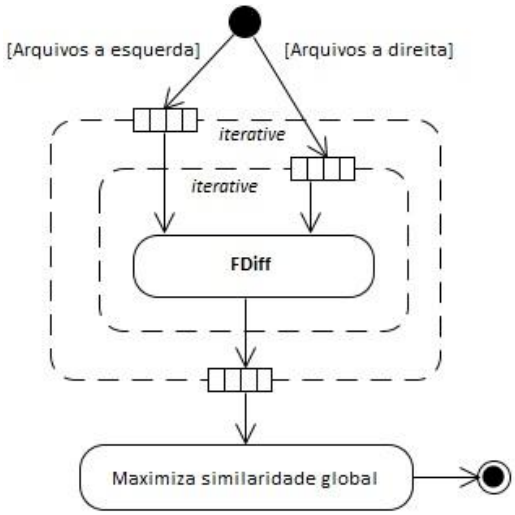


Figura 26 – Diagrama de atividades – Processa arquivos similares

Todos os arquivos restantes são considerados como mudanças ao longo do mesmo arquivo, mesmo se o nome ou *path* sejam diferentes. Ou seja, todos os arquivos restantes são considerados como mudanças, independentemente da estrutura de subdiretórios. As listas de arquivos da esquerda e da direita contêm elementos com um determinado path, nome e conteúdo, porém a análise é toda baseada no conteúdo, independentemente do *path* e nome.

Após esta etapa, todos os arquivos de texto foram combinados em $O(n^3)$. Vale ressaltar que, de acordo com Estublier (2000), duas revisões sucessivas são em média 98% similares. Com isso, $N \gg n$ e $M \gg m$, reduzindo o impacto desta etapa no desempenho global do algoritmo.

Um efeito colateral positivo do método é que a produção da matriz de similaridades, utilizada para detectar a combinação ótima descrita no passo anterior, permite a identificação de todos os arquivos que possuam alguma semelhança com um determinado arquivo. Esta informação é útil para ajudar na identificação de refatorações que transcendem as fronteiras de arquivos.

3.3.2 **FDIFF**

A comparação pareada entre arquivos direta, ao selecionar dois arquivos, ou indireta, ao executar o DDiff, ocorre com a utilização do algoritmo de programação dinâmica que detecta o LCS entre os arquivos em $O(n_f \times m_f)$ onde m_f e n_f representam o número de linhas existentes nos arquivos comparados. A Figura 27 apresenta o diagrama de atividades

referente ao módulo FDiff, indicando a execução do algoritmo LCS e a redução da granularidade utilizada durante a análise dos arquivos.

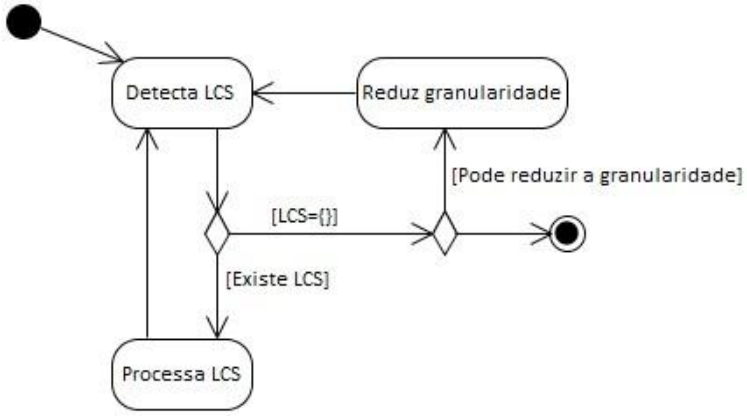


Figura 27 – Diagrama de atividades - FDiff

A primeira execução algoritmo LCS identifica as similaridades conforme a maioria das ferramentas existentes, ou seja, a maior sequência comum entre os arquivos é detectada na granularidade base da ferramenta. A maior parte das ferramentas existentes utiliza o grão linha como base, porém algumas fazem uso do grão caractere para a detecção das similaridades do conteúdo de arquivos, o que pode provocar problemas em termos de performance, uma vez que aumenta consideravelmente o número de entradas para o algoritmo de comparação. Na proposta apresentada neste trabalho, o processo de identificação de diferenças e similaridades é iniciado no grão linha e, conforme as todas sequências comuns forem encontradas, o grão é refinado e o algoritmo é reexecutado. A Figura 28 exemplifica a primeira execução do LCS no grão linha para comparação de duas versões da classe *PagamentoCheque.java* descritas no exemplo motivacional da seção anterior. Para melhor visualização do resultado obtido pela execução do algoritmo LCS a sequência comum encontra-se destacada nas figuras a seguir.

Depois da identificação do LCS, o algoritmo remove a sequência de ambos os arquivos e recomeça a detecção, de forma iterativa, até que nenhuma subsequência comum seja encontrada para a granularidade utilizada. Da segunda execução em diante, FDiff é capaz de encontrar sequências comuns que representam blocos em movimento no código.

O objetivo principal é detectar todas as possíveis similaridades entre os arquivos comparados considerando a mesma granularidade (i.e., linha). Depois disso, o grão é refinado e o algoritmo é executado novamente. Este algoritmo foi denominado no contexto deste trabalho como ILCS, do inglês, iterative LCS. A Figura 29 exemplifica o funcionamento do

algoritmo, considerando o exemplo já exposto, com a utilização do grão palavra, depois de esgotar as sequências encontradas no grão linha.

```
public class PagamentoCheque extends Pagamento {
  public Integer numeroParcelas;
}
```

(a) Arquivo à esquerda

```
public class PagamentoCheque extends Pagamento {
  private Integer numeroParcelas;
  public void getNumeroParcelas() {
    return this.numeroParcelas;
  }
  public void setNumeroParcelas(String numeroParcelas) {
    this.numeroParcelas = numeroParcelas;
  }
}
```

(b) Arquivo à direita

Figura 28 – Execução do algoritmo de LCS no grão linha

Na sequência de execução, o grão é refinado para caractere e segue os mesmo passos descritos anteriormente na transição do grão linha para o grão palavra. No exemplo exposto o grão palavra esgota todas as possibilidades de detecção de semelhanças, ou seja, a execução no grão caractere retorna $LCS = \{\emptyset\}$.

Neste ponto, é importante salientar a complexidade de processamento do ILCS. Para caracterizar a importância da redução de granularidade, para diminuição do tempo de processamento, é exposto um exemplo simplificado:

Supondo a comparação de dois arquivos contendo 100 linhas cada, com 50 caracteres em cada linha, teríamos um processamento no grão linha de $100^2 = 10.000$, considerando a complexidade assintótica do LCS de $O(n^2)$. Se o processamento iniciasse com o grão caractere teríamos $(100 \times 50)^2 = 25.000.000$.

Sendo assim, teríamos uma proporção de $\frac{10.000}{25.000.000} = \frac{1}{2.500}$ caso o IDIFF fizesse o cálculo diretamente a partir do grão caractere, sem redução de granularidade. Diante disso, a decisão de realizar o refinamento do grão apenas após esgotar as sequências encontradas foi tomada para proporcionar um tempo de processamento satisfatório, principalmente durante a análise de diretórios, onde o número de arquivos pode ser extenso.

```
public Integer numeroParcelas;
```

(a) Arquivo à esquerda

```
private Integer numeroParcelas;
public void getNumeroParcelas() {
  return this.numeroParcelas;
}
public void setNumeroParcelas(String numeroParcelas) {
  this.numeroParcelas = numeroParcelas;
}
```

(b) Arquivo à direita

Figura 29 – Execução do algoritmo de LCS no grão palavra

3.3.3 **VISUALIZAÇÃO DE RESULTADOS**

A visualização dos resultados depende da perspectiva escolhida pelo usuário. IDIFF oferece a visualização de diferenças e de similaridades. A primeira opção é destinada a situações habituais, em que ambos os arquivos são semelhantes, com pequenas diferenças. A Figura 30 mostra esta visualização para a comparação proposta no exemplo motivacional.

```
public class PagamentoCheque extends Pagamento {
  public Integer numeroParcelas;
}
```

(a) Arquivo à esquerda

```
public class PagamentoCheque extends Pagamento {
  private Integer numeroParcelas;
  public void getNumeroParcelas() {
    return this.numeroParcelas;
  }
  public void setNumeroParcelas(String numeroParcelas) {
    this.numeroParcelas = numeroParcelas;
  }
}
```

(b) Arquivo à direita

Figura 30 – Visualização de diferenças entre duas versões de uma classe

Por outro lado, a perspectiva de similaridades é útil para situações em que os dois arquivos diferem substancialmente. No geral, a visualização de diferenças é indicada para o resultado ótimo gerado pelo algoritmo Húngaro. Para as demais indicações, a visualização de similaridades apresenta resultados satisfatórios.

A Figura 31 mostra a comparação entre as duas versões da classe *PagamentoCheque.java*, através da perspectiva de similaridades. Esta perspectiva se mostrou mais apropriada, pois a classe possuía apenas um atributo e nenhum trecho de código. Se fosse uma classe real, com centenas de linhas de código, a aplicação da refatoração *Encapsulate Field* provavelmente levaria ao uso de visualização de diferenças.

FDiff também é capaz de exibir as diferenças e semelhanças de um arquivo específico em relação aos outros arquivos existentes. Isto permite a visualização de todas as mudanças que afetaram um determinado arquivo. A Figura 32 representa a identificação de similaridades encontradas entre a classe *PagamentoCheque.java* e as classes *PagamentoCheque.java* e *PagamentoCartao.java*.

```
public class PagamentoCheque extends Pagamento {
  public Integer numeroParcelas;
}
```

(a) Arquivo à esquerda

```
public class PagamentoCheque extends Pagamento {
  private Integer numeroParcelas;
  public void getNumeroParcelas() {
    return this.numeroParcelas;
  }
  public void setNumeroParcelas(String numeroParcelas) {
    this.numeroParcelas = numeroParcelas;
  }
}
```

(b) Arquivo à direita

Figura 31 – Visualização de semelhanças entre duas versões de uma classe

```
public class PagamentoCheque extends Pagamento {
      public Integer numeroParcelas;
}
```

```
public class PagamentoCheque extends Pagamento {
      private Integer numeroParcelas;
      public void getNumeroParcelas() {
            return this.numeroParcelas;
      }
      public void setNumeroParcelas(String numeroParcelas) {
            this.numeroParcelas = numeroParcelas;
      }
}
```

```
public class PagamentoCartao extends Pagamento {
      public Integer numeroParcelas;
}
```

Figura 32 – Análise de diferenças entre arquivos

3.4 **CONSIDERAÇÕES FINAIS**

Este capítulo apresentou uma abordagem para detecção de diferenças baseada em conteúdo, capaz de identificar semelhanças que transcendem os limites de um arquivo. A preocupação constante em evitar um tempo de processamento excessivo auxiliou na construção das principais características da abordagem. Todos os passos existentes utilizam algoritmos que propiciam a redução do tempo de processamento e a realização de detecção precisa das diferenças existentes entre versões de um artefato. Conforme dito anteriormente, a principal motivação é a possibilidade de identificar alterações motivadas por refatoração de código, permitindo assim uma melhor compreensão das alterações realizadas.

Capítulo 4 – **FERRAMENTA IDIFF**

4.1 **INTRODUÇÃO**

A abordagem apresentada no Capítulo 3 foi implementada na linguagem Java, em uma ferramenta homônima baseada no framework Swing para interação gráfica com o usuário, ambos escolhidos por conhecimento prévio, facilitando o desenvolvimento do protótipo. O processo de cálculo de hash utiliza o algoritmo SHA-1[5], do inglês *Secure Hash Algorithm*. De forma sucinta, este algoritmo recebe como entrada uma mensagem de tamanho arbitrário e produz uma impressão digital única de $160 - bit$ por arquivo. Apesar da escolha do SHA1, por ser considerado mais seguro, com probabilidade reduzida de conflitos, o algoritmo de hash é um ponto de extensão desse trabalho, permitindo a troca por outras implementações, se desejado. No que diz respeito a implementação, a classe *MessageDigest* da biblioteca *java.security*[6] foi utilizada para cálculo de hash durante a análise de arquivos idênticos.

Além disso, a implementação do algoritmo Húngaro, desenvolvida por Nedas[7], foi utilizada para identificação da similaridade global ótima na comparação de diretórios. Por fim, o algoritmo do módulo FDiff tem sua implementação baseada no LCS com aplicação de programação dinâmica, de acordo com Cormen et. al. (2009).

Este capítulo apresenta a implementação realizada para concretização da abordagem. As subseções a seguir apresentam a execução da ferramenta usando exemplo motivacional do Capítulo 3 como base, com o objetivo de exibir as principais funcionalidades disponíveis e as restrições da implementação. A utilização da ferramenta pode ser dividida em três fases principais:

- **Configuração da ferramenta:** descrita na seção 4.2, a configuração pode ser realizada de forma a atender as necessidades do usuário. A ferramenta possui uma configuração padrão baseada nos melhores resultados obtidos nos experimentos realizados, apresentados no Capítulo 5.

- **Análise de diretórios:** A seção 4.3 apresenta detalhes sobre a análise de diretórios no módulo DDiff, além de descrever o funcionamento e a interpretação dos resultados.

[5] http://www.itl.nist.gov/fipspubs/fip180-1.htm
[6] http://docs.oracle.com/javase/1.4.2/docs/api/java/security/MessageDigest.html
[7] http://konstantinosnedas.com/dev/soft/munkres.htm

- **Análise de arquivos:** A seção 4.4 descreve o a execução do módulo FDiff da ferramenta e apresenta exemplos com o intuito de expor resultados de acordo com a configuração da ferramenta.

Finalmente a seção 4.5 apresenta as considerações finais indicando as vantagens da utilização da ferramenta e os resultados obtidos pela sua utilização.

4.2 CONFIGURAÇÃO DA FERRAMENTA

Inicialmente o IDIFF apresenta uma tela para informação dos artefatos a serem comparados, conforme pode ser visto na Figura 33. Diretórios e arquivos textos são aceitos para a execução, de forma que o primeiro artefato informado é considerado versão base e o segundo a revisão da versão indicada.

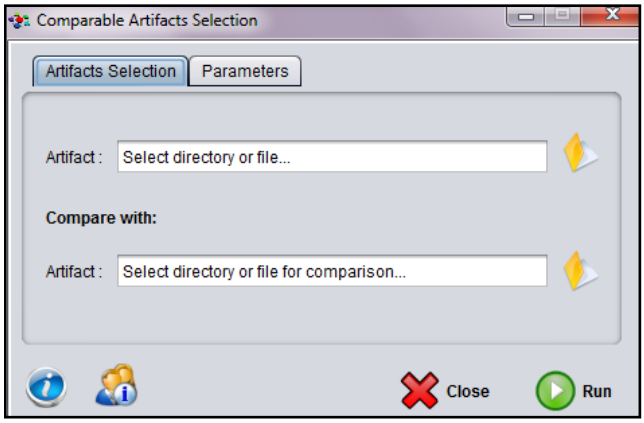


Figura 33 – Seleção de artefatos comparáveis

O próximo passo a ser realizado na ferramenta IDIFF para comparação de artefatos é o ajuste de parâmetros a serem utilizados durante a execução. Existem três diferentes tipos de parâmetros: granularidade, separadores e limiar, conforme pode ser visto na Figura 34**.**

A parametrização da granularidade indica o grão mais fino que será utilizado pelo algoritmo ILCS. Ou seja, se o grão “linha” for selecionado, o algoritmo não irá refinar a granularidade durante sua execução e apenas este grão será considerado para análise das diferenças. Por sua vez, se o grão “palavra” for escolhido, a análise será inicialmente realizada com o grão “linha” e, após esgotar todas as sequências possíveis neste grão, a ferramenta refinará a granularidade para palavra e buscará todas as subsequências comuns com este grão. O mesmo acontece para o grão “caractere”.

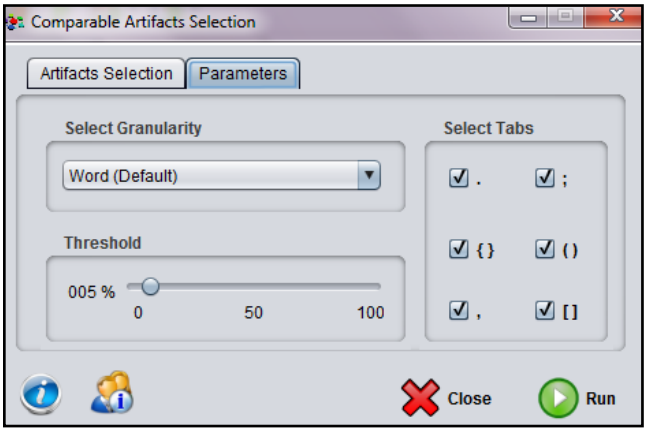


Figura 34 – Seleção de parâmetros do algoritmo

Em seguida, a parametrização de separadores deve ser realizada, com intuito de traduzir para o cenário da programação os separadores de palavras. Para melhor compreensão suponha o método *main* apresentado na Figura 35; sem a utilização de separadores de palavra o trecho demarcado é considerado como uma única palavra, quando o ideal seria a detecção de quatro palavras: *System*, *out*, *println* e *"Separadores"*.

```
public static void main(String[] args){
  System.out.println("Separadores");
}
```

Figura 35 – Exemplo de separadores de palavras

Finalmente, a parametrização de limiar identifica o menor grau de similaridade a ser considerado durante a comparação de arquivos no módulo DDiff. É importante lembrar que nos cenários analisados os artefatos em questão são códigos escritos em uma mesma linguagem, que seguem uma gramática. Desta forma, é comum haver algum grau de similaridade entre os arquivos mesmo sendo arquivos essencialmente diferentes. Com isso, essa parametrização visa limitar a identificação destas similaridades provenientes da gramática utilizada.

Esta parametrização apresenta o seguinte efeito durante a execução da ferramenta: se o limiar é de 5%, apenas similaridades superiores a este percentual serão identificadas. As demais serão marcadas como deleção e adição, para o diretório à esquerda e à direita, respectivamente. Os percentuais de similaridades encontradas entre os arquivos são indicados

pela ferramenta após a seleção de arquivos comparáveis, conforme pode ser visto na seção 4.3.

4.3 **DDIFF - ANÁLISE DE DIFERENÇAS ENTRE DIRETÓRIOS**

A Figura 36 e a Figura 37 apresentam visualizações da análise pareada de duas versões de um mesmo diretório. A legenda, na parte superior da tela, indica o significado de cada cor utilizada na apresentação de resultados. Além disso, o símbolo asterisco (∗) identifica a existência de alterações no interior de um diretório, como deleção, adição, movimentação ou edição.

Para ilustrar a identificação de arquivos incluídos e excluídos na análise de diretórios, duas classes foram adicionadas ao código do exemplo motivacional, sendo elas *HistoricoCompra.java* e *HistoricoVenda.java*. A *HistoricoCompra.java* foi removida do diretório à esquerda, versão base, e *HistoricoVenda.java* foi adicionada no diretório à direita, versão refatorada. A Figura 36 ilustra a execução da ferramenta no grão "linha". Neste cenário, também é possível notar a identificação da movimentação de um arquivo de configuração *Catalogo.xml*. A ferramenta é capaz de identificar esta alteração, pois detecta que o conteúdo do arquivo é idêntico, mesmo com o nome completo (i.e., *path* + nome do arquivo) sendo diferente.

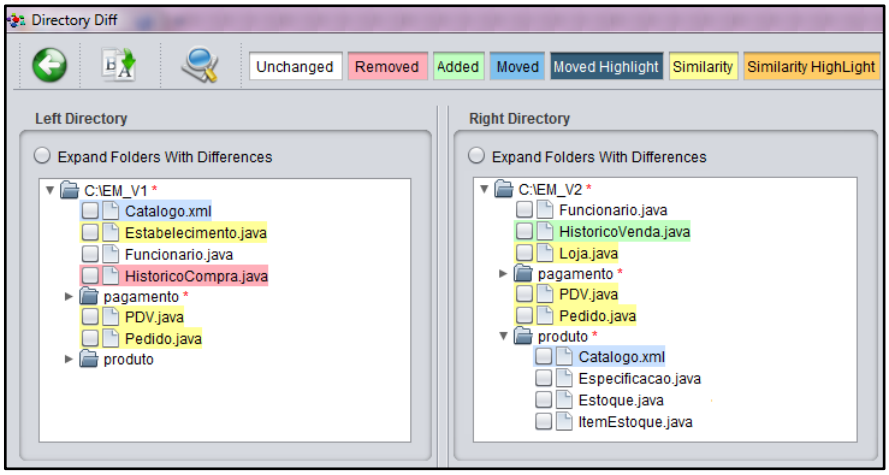


Figura 36 – Visualização de arquivos movidos, incluídos, excluídos e similares

Na comparação de diretórios, cada arquivo é comparado com todos os outros arquivos do diretório oposto, exceto aqueles não comparáveis (arquivos binários) e os já identificados como idênticos, renomeados ou movidos (ou seja, mesmo conteúdo, mas com possíveis

mudanças no nome ou no *path*). O grau de similaridade exibido é calculado a partir dos resultados obtidos pela execução do algoritmo ILCS através do módulo FDiff da ferramenta.

A Figura 37 apresenta a comparação de diretórios e a seleção da classe *Estabelecimento.java*. Essa seleção exibe vários pontos importantes desenvolvidos neste trabalho. Inicialmente é possível observar que a classe *Loja.java* apresentou 91% de similaridade com a classe selecionada. Além disso, esta classe é indicada como a melhor escolha (*Best Choice*) do algoritmo Húngaro. É importante lembrar que a escolha do Húngaro visa otimizar a similaridade global (i.e., dos diretórios) durante a comparação.

Ainda nesta figura, é possível visualizar as classes que apresentaram similaridade inferior ao limiar parametrizado na ferramenta. Nesta situação os arquivos são identificados como exclusão (diretório à esquerda) ou inclusão (diretório à direita). Diferentemente dos arquivos que não possuem similaridades com nenhum outro arquivo, como, por exemplo, as classes *HistoricoCompra.java* e *HistoricoVenda.java*, as classes ignoradas pelo limiar apresentam a indicação do percentual de similaridade encontrado e permite que o usuário realize a comparação pareada dos arquivos.

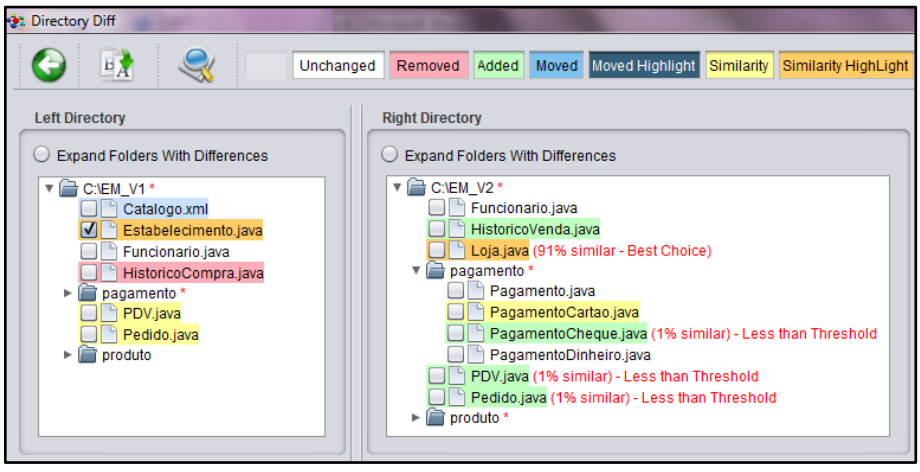


Figura 37 – Comparação entre duas versões de um diretório

Independente da granularidade escolhida para execução, o percentual é calculado como apresentado anteriormente.

A partir deste cálculo, o algoritmo Húngaro identifica a similaridade ótima global, onde nem sempre o arquivo com maior grau de similaridade local é informado como melhor escolha. A Figura 38 ilustra este cenário, onde a classe *PagamentoCheque.java* possui 40% de similaridade com a classe *PagamentoCartao.java* e 30% de similaridade com *PagamentoCheque.java*. Apesar da classe *PagamentoCartao.java* possuir um maior grau de

similaridade, a outra classe foi escolhida como melhor escolha do algoritmo Húngaro, otimizando assim o resultado global da comparação de todos os arquivos do diretório.

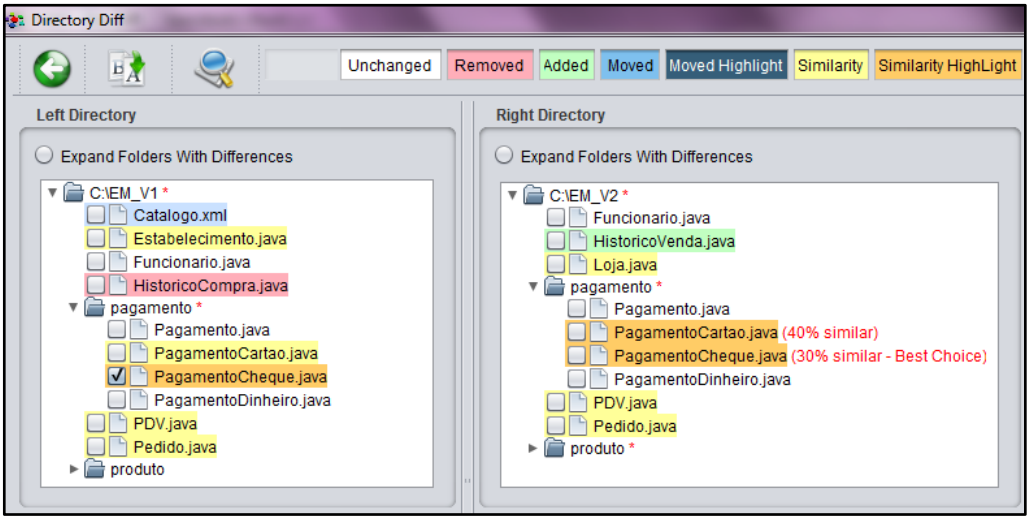


Figura 38 - Verificação de similaridade global detectada pelo algoritmo Húngaro

A Tabela 12 apresenta um quadro com os valores de todos os possíveis casamentos entre os arquivos do diretório à esquerda e os da direita. As linhas representam os arquivos selecionados, pertencentes ao diretório à esquerda. As colunas representam os arquivos comparados com cada um dos arquivos selecionados, pertencentes ao diretório à direita. Os respectivos valores de similaridades são apresentados nas células da tabela.

Tabela 12 – Similaridades ótima global x similaridade ótima local

	Arquivos similares →				
↓ Arquivos selecionados	**Loja.java**	**PagamentoCartao.java**	**PagamentoCheque.java**	**PDV.java**	**Pedido.java**
Estabelecimento.java	**91**	0	1	1	1
PagamentoCartao.java	0	**60**	20	0	0
PagamentoCheque.java	0	**40**	30	0	0
PDV.java	0	0	0	**73**	66
Pedido.java	1	1	1	1	**30**

As células destacadas na cor amarela indicam as escolhas do Húngaro. Por sua vez, os números em vermelho representam as similaridades ótimas locais para cada arquivo selecionado. O somatório das similaridades detectadas como melhor escolha do Húngaro é $(91 + 60 + 30 + 73 + 30) = 284$, que otimiza a similaridade global.

4.4 **FDIFF – ANÁLISE DE DIFERENÇAS ENTRE ARQUIVOS**

A partir da tela de comparação de diretórios é possível realizar a seleção de arquivos para comparação pareada ou múltipla. É importante destacar que o módulo DDiff só permite a comparação de arquivos que possuem algum grau de similaridade, mesmo que este seja inferior ao limite selecionado. A seguir, são apresentadas essas duas formas de comparação de arquivos.

4.4.1 **COMPARAÇÃO PAREADA**

A comparação pareada envolve dois arquivos selecionados no módulo DDiff, conforme pode ser visto na Figura 39. A comparação baseada em conteúdo considera todos os parâmetros informados na configuração da ferramenta para apresentação de resultados. Esta comparação exemplifica a identificação de arquivos renomeados. Assim como no módulo DDiff, a legenda de cores indica quais foram as ações realizadas no conteúdo destacado.

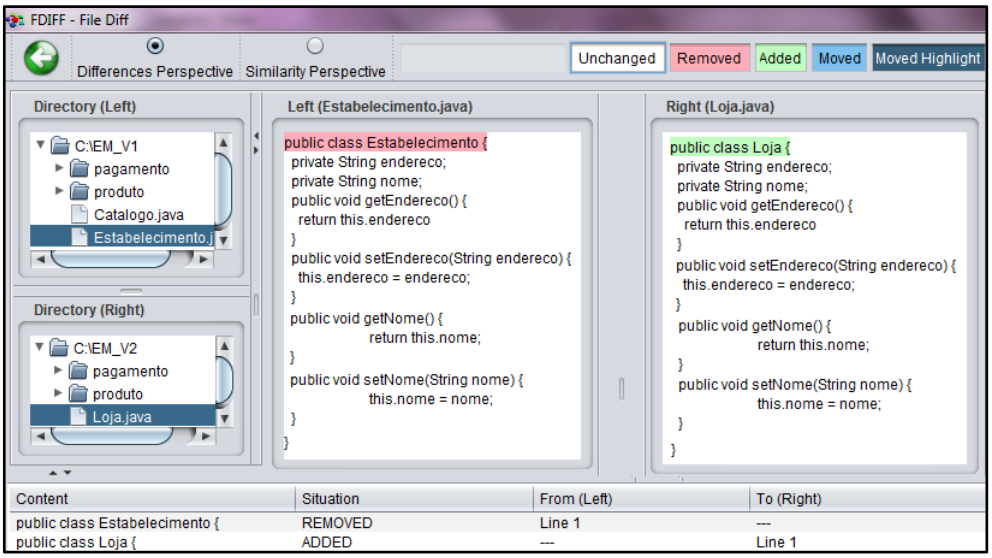


Figura 39 – Comparação de arquivos visualizados na perspectiva de diferenças

A tela do FDiff permite ao usuário visualizar os resultados em termos de duas diferentes perspectivas: perspectiva de diferenças e perspectiva de similaridade. O IDIFF seleciona automaticamente a perspectiva mais relevante, por padrão. Para isso, ele considera o nível de semelhança, conforme detalhado na Figura 40. O exemplo exposto na Figura 39 apresenta o resultado na perspectiva de diferenças, pois os arquivos em questão possuem 91% de similaridade.

$$Grau\ de\ similaridade \geq 50\% \rightarrow Perspectiva\ de\ diferenças$$
$$Grau\ de\ similaridade < 50\% \rightarrow Perspectiva\ de\ similaridade$$

Figura 40 – Percentual para identificação de perspectiva do IDIFF

A análise de perspectivas tem como principal objetivo evitar a poluição dos resultados exibidos, permitindo foco nas alterações relevantes. Como exemplo, podemos retomar a um dos cenários do exemplo motivacional exposto para este trabalho. Durante a análise da movimentação de métodos, duas classes diferentes são avaliadas. Em nosso exemplo, o método *paga(Integer idPedido)* foi movido da classe *PDV.java* para a classe *Pedido.java*, conforme exibido na Figura 41.

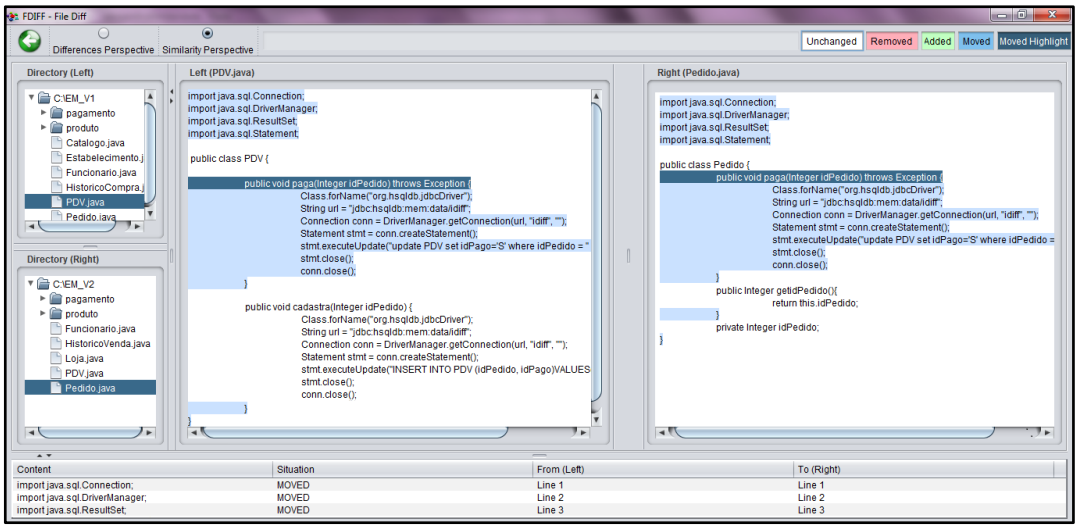


Figura 41 – Perspectiva de similaridades considerando grão linha

Diante disso, a comparação na perspectiva de diferenças torna a análise mais difícil, uma vez que a maior parte do conteúdo dos arquivos difere. Por se tratar de arquivos distintos, nesta situação o ideal é utilizar a perspectiva de similaridade, que irá indicar a movimentação realizada sem considerar o que difere entre os arquivos. Além disso, eventos de mouse permitem ao usuário identificar de forma exata a posição do conteúdo movido.

4.4.2 **COMPARAÇÃO MÚLTIPLA**

Na Figura 37, se apenas um arquivo é selecionado na tela, o sistema apresenta a comparação múltipla de arquivos. Esta comparação permite a identificação de semelhanças

entre um arquivo de um diretório e todos os outros arquivos do segundo diretório em análise, Todas as semelhanças são exibidas em uma única tela, como mostrado na Figura 42.

Uma observação importante a ser feita está relacionada com a detecção de similaridades em um mesmo trecho de código em duas ou mais classes. Neste cenário, o trecho com maior percentual de similaridade possui prioridade na exibição dos resultados. Os demais arquivos terão a similaridade apresentada com a aplicação do filtro referente ao arquivo em questão. A Figura 42 apresenta esta situação, onde os *imports* são idênticos para duas classes detectadas com similaridades. Desta forma, como o percentual de similaridade com a classe *PDV.java* é maior, esta é exibida ao passar o mouse sobre a linha comparada.

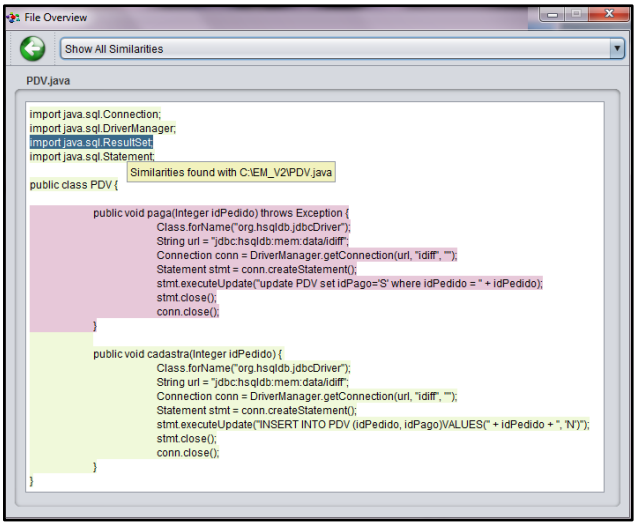


Figura 42 – Comparação múltipla da classe PDV.java

A utilização esperada dessa visualização é apoiar na detecção de refatorações que transcendem a fronteira de um arquivo, como, por exemplo, a movimentação de métodos entre classes. A visualização em questão permite filtrar os arquivos a serem analisados a partir de uma combo disponível na parte superior da mesma. No exemplo exposto, foram encontradas similaridades com duas classes, *PDV.java* e *Pedido.java*, ambas disponíveis para filtragem de resultados, conforme Figura 43.

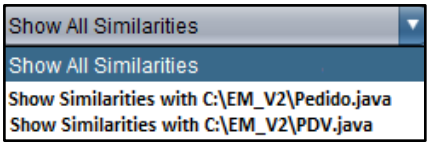


Figura 43 – Filtro disponível na comparação múltipla da classe PDV.java

Após a aplicação do filtro, o sistema exibe similaridade conforme exibido na Figura 44. Em todas as telas já apresentadas, todos os resultados são realçados por meio de cores e eventos de mouse, com o objetivo de melhorar a usabilidade da ferramenta e tornar a compreensão dos resultados mas simples e agradável para o usuário.

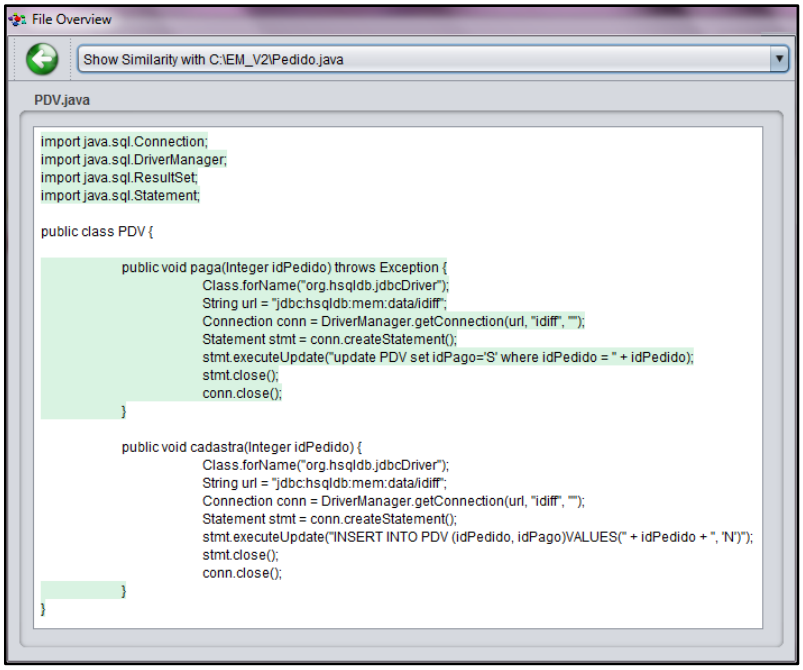


Figura 44 – Comparação múltipla da classe PDV.java com aplicação de filtro

4.5 **CONSIDERAÇÕES FINAIS**

Este capítulo apresentou detalhes sobre a ferramenta desenvolvida para comparação de diretórios e arquivos, conforme a proposta deste trabalho. É importante lembrar que o IDIFF foi desenvolvido para facilitar a compreensão das alterações realizadas, detectando de forma específica o que foi mudado, sem identificar a ação que levou àquela mudança. Ou seja, a ferramenta é capaz de detectar um cenário de refatoração mas não identifica qual refatoração foi realizada. Isso ocorre devido ao fato da ferramenta não ser especializada em linguagem de programação, não realizando assim análises sintáticas ou semânticas.

Existem pontos positivos e negativos nesta abordagem. O fato de não ser especializado em uma linguagem de programação específica permite a utilização da ferramenta sobre código escrito em qualquer linguagem. Por outro lado, devido a esta generalização, não é possível retornar ao usuário resultados mais específicos, como, por exemplo, a identificação

de qual refatoração foi aplicada ou desconsiderar semelhanças relacionadas a gramática utilizada, o que facilitaria a visualização, por omitir resultados com informações irrelevantes.

Capítulo 5 – **AVALIAÇÃO**

5.1 **INTRODUÇÃO**

A principal motivação deste capítulo é responder às seguintes questões de pesquisa, definidas no Capítulo 1:

- IDIFF aumenta a precisão quando comparado a ferramentas de *diff* convencionais?
- IDIFF aumenta a cobertura, quando comparado a ferramentas de *diff* convencionais?
- Para quais tipos de refatoração o IDIFF possui resultados mais precisos?
- Qual é a parametrização de granularidade para IDIFF que apresenta resultados mais precisos?

Para a escolha da ferramenta de diff convencional a ser utilizada como *baseline*, foi realizado um *survey*, descrito no Apêndice B, com objetivo de identificar características fundamentais e desejáveis em ferramentas capazes de identificar diferenças em artefatos textuais, de forma a identificar o estado da prática. O resultado do *survey* mostrou que a ferramenta WinMerge é a mais utilizada pelos respondentes e gera maior satisfação (resposta "excelente") dentre as indicadas. Desta forma, WinMerge foi escolhida.

Para avaliar a qualidade dos resultados apresentados pelo IDIFF, em comparação ao WinMerge, foi necessário aplicá-los sobre diferentes cenários de modificação de artefatos, em especial os que envolvem refatorações. Desta forma, setenta e seis refatorações diferentes contidas no catálogo do Fowler (1999) foram escolhidas, juntamente com exemplos que retratam o resultado esperado de cada refatoração. Esses resultados esperados são tratados como gabaritos no contexto da avaliação. Essa escolha se deve à ampla utilização dessas refatorações, e pelo livro do Fowler ser considerado a principal referência no tema, possibilitando assim a construção de dados imparciais para avaliação das ferramentas.

Por fim, para comparar a qualidade dos resultados apresentados pelo IDIFF e pelo WinMerge, foram utilizadas três métricas: Precisão, Cobertura e Aderência (média harmônica entre Precisão e Cobertura). A primeira métrica tem como intuito verificar a corretude das respostas. A Cobertura, por outro lado, verifica a completude. Já a Aderência combina ambas as métricas anteriores, de forma a identificar conjuntamente o quanto correto e completo está

o resultado se comparado ao gabarito. O fato das métricas precisão e cobertura serem de grandezas inversamente proporcionais influenciou na escolha de média harmônica para o cálculo da aderência durante os experimentos.

O restante deste capítulo está organizado em cinco seções além dessa introdução. A seção 5.2 descreve detalhes sobre o planejamento dos experimentos. A execução do experimento é explicada na seção 5.3. A análise estatística é apresentada na seção 5.4, detalhando testes realizados, seus respectivos resultados e conclusões sobre os dados obtidos. Ameaças à validade são apresentadas na seção 5.5. Finalmente a seção 5.6 apresenta as considerações finais deste capítulo.

5.2 **PLANEJAMENTO DO EXPERIMENTO**

Inicialmente foi realizado um levantamento sobre ferramentas de *diff* através de um *survey*, disponível no Apêndice B. Sessenta e três pessoas responderam perguntas relacionadas ao grau de satisfação na utilização de ferramentas capazes de detectar diferenças em diversos tipos de artefatos. Indicando as principais características, essenciais e desejáveis, para ferramentas de *diff*. Dentre os participantes, existem pessoas com experiência em projetos de indústria (79%) e com experiência em projetos acadêmicos (56%). Nesta pesquisa, o WinMerge[8] uma ferramenta de *diff Open Source* para *Windows*, obteve o maior percentual de satisfação na sua utilização: cerca de 90% (respostas "excelente" e "boa"), considerando os usuários da ferramenta. Por esta razão, o WinMerge foi selecionado como *baseline* para comparação com IDIFF. Esta ferramenta possibilita a parametrização de granularidades além da separação de palavras por espaços em branco e pontuação, conforme pode ser visto na Figura 45.

Neste experimento são utilizados sete diferentes grupos de refatorações, definidos por Fowler (1999), totalizando setenta e seis tipos de refatorações analisadas, também definidas por Fowler (1999). Vinte e duas refatorações foram descartadas devido à ausência de dados suficientes para construção de exemplos e gabaritos para execução dos experimentos. A listagem completa das refatorações encontra-se no Apêndice D.

O livro do Fowler apresenta, além da discussão teórica sobre as refatorações, exemplos que demonstram os estados antes e após a aplicação das refatorações. Estes exemplos foram utilizados para construção dos arquivos e empregados na avaliação das

[8] http://winmerge.org/

ferramentas. Neste trabalho todos os arquivos de entrada foram construídos na linguagem Java e apenas códigos descritos no livro foram introduzidos nos exemplos.

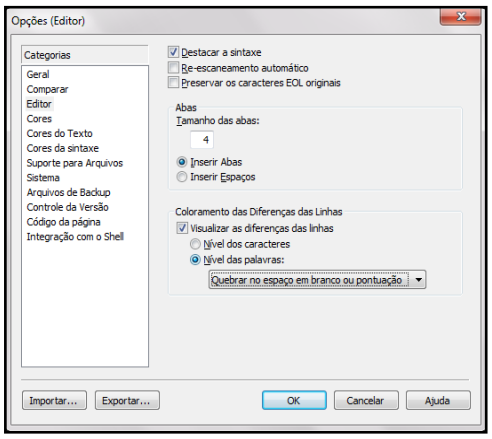


Figura 45 – Parametrização da ferramenta WinMerge

No que diz respeito ao gabarito, o mesmo também é extraído do livro, pois todos os passos para execução da refatoração são detalhados pelo autor. Este gabarito é definido como forma de indicar o resultado que melhor representa cada refatoração, em termos de operações passíveis de execução, ou seja, adição, deleção, modificação e movimentação. Desta forma as seguintes variáveis independentes foram definidas para o experimento:

- Granularidade
- Tags (separadores de palavras)
- Limites

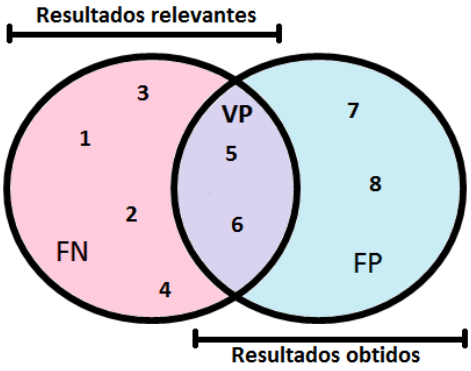
Os arquivos de entrada são executados três vezes em cada ferramenta, uma para cada granularidade: linha, palavra e caractere. As demais parametrizações foram utilizadas conforme configuração padrão das ferramentas.

Para cada execução é armazenada a quantidade de resultados verdadeiros positivos (VP), que representam os valores detectados corretamente, falsos positivos (FP), que indicam os resultados não esperados, e falsos negativos (FN), para os resultados não detectados. Todos contabilizados pelo número de caracteres.

A comparação de resultados é realizada em termos das operações de adição, deleção, modificação e movimentação, conforme gabarito. Estes dados são utilizados para cálculo das seguintes variáveis dependentes do experimento:

- Precisão
- Cobertura
- Aderência

Precisão, Cobertura e Aderência (BAEZA-YATES; RIBEIRO-NETO, 1999) são métricas tradicionais para recuperação de informação. A Precisão pode ser vista como uma medida de corretude, ou seja, percentual de resultados obtidos que são relevantes. Por outro lado, a Cobertura é uma medida de completude, que verifica percentual de resultados relevantes que foram obtidos. Por fim, a Aderência corresponde à média harmônica dos resultados obtidos de Precisão e Cobertura, visando um balanceamento entre as duas métricas. A Figura 46 exemplifica as métricas citadas anteriormente,



$$Precisão = \frac{VP}{VP + FP}$$

$$Cobertura = \frac{VP}{VP + FN}$$

$$Aderência = 2 \times \frac{Precisão \times Cobertura}{Precisão + Cobertura}$$

Figura 46 – Precisão, cobertura e aderência

De acordo com exemplo exposto:

Resultados relevantes = {1,2,3,4,5,6} Resultados obtidos = {5,6,7,8}

$\text{Precisão} = \frac{2}{2+2} = 50\%$ $\text{Cobertura} = \frac{2}{2+4} \cong 34\%$

$$Média\ Harmônica = 2 \times \frac{\left(\frac{2}{4} \times \frac{2}{6}\right)}{\left(\frac{2}{4} + \frac{2}{6}\right)} = 2 \times \frac{1}{5} = 40\%$$

Após obtenção das variáveis do experimento é iniciada a realização de testes de hipóteses, visando a comparação entre os resultados das ferramentas. o Teste de Hipótese é o processo de inferir sobre uma população a partir de uma amostra, de tal forma que seja

possível verificar a compatibilidade dos dados obtidos com a hipótese definida. A hipótese é uma afirmação sobre esta população que será avaliada através de procedimentos estatísticos (SPIEGEL, 1994).

Outro fator importante para o planejamento do experimento diz respeito a definição do nível de significância a ser utilizado para cálculos das estatísticas. Para os experimentos realizados neste trabalho foi utilizado um intervalo de confiança de 95%, ou seja $\alpha = 0{,}05$, onde α é a probabilidade máxima de rejeitar uma hipótese nula verdadeira.

5.3 **EXECUÇÃO DO EXPERIMENTO**

A execução do experimento seguiu o plano detalhado na seção anterior. O processo de execução é apresentado no diagrama de atividades da Figura 47. Como pode ser observado, inicialmente são obtidos os arquivos de entrada (antes e depois da refatoração) e o arquivo de gabarito, que indica as ações que foram de fato feitas durante a refatoração. Os arquivos de entrada são processados por ambas as ferramentas e os resultados das ferramentas são comparados com o gabarito. Essa comparação visa identificar o quanto as ferramentas foram capazes de recuperar as ações que de fato foram feitas durante a refatoração. Para avaliar isso, são calculadas as métricas de Precisão, Cobertura e Aderência.

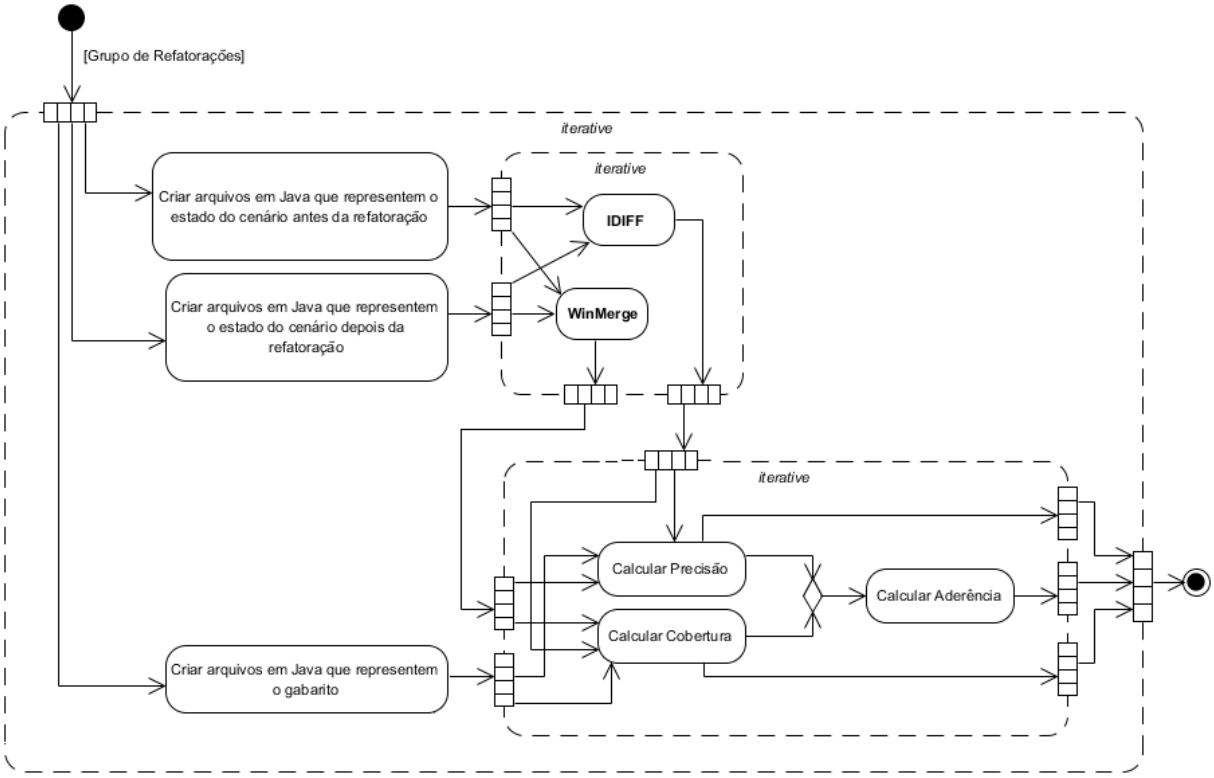


Figura 47 – Diagrama de atividades da execução do experimento

Para ilustrar um ciclo de execução do experimento de forma didática, a refatoração denominada *Inline Method*, pertencente ao grupo *Composing Methods*, foi escolhida. Esta refatoração pode ser utilizada quando o conteúdo do método é de fácil compreensão, assim como seu nome, ou quando um código está mal estruturado e todos os métodos são incluídos em um único método para posterior reestruturação. A refatoração consiste em substituir a chamada do método pelo seu conteúdo e em seguida excluir o método. A Figura 48 exemplifica um trecho da primeira versão do arquivo a ser comparada, e, em seguida, a Figura 49 apresenta o código após a aplicação da refatoração *Inline Method.* Os conteúdos dos exemplos expostos nestas figuras foram extraídos de Fowler (1999).

```
public class Main {
      int getRating() {
            return (moreThanFiveLateDeliveries()) ? 2 : 1;
      }
      boolean moreThanFiveLateDeliveries() {
            return _numberOfLateDeliveries > 5;
      }
}
```

Figura 48 – Versão base do arquivo de entrada

```
public class Main {
      int getRating() {
            return (_numberOfLateDeliveries > 5) ? 2 : 1;
      }
}
```

Figura 49 – Versão refatorada do arquivo de entrada

A Tabela 13 apresenta o gabarito referente ao exemplo exposto. Nela é possível notar a presença de todos os passos realizados durante a refatoração, como, por exemplo, a exclusão do conteúdo e da chamada do método *moreThanFiveLateDeliveries()* e a movimentação de *_numberOfLateDeliveries > 5.*

Após a obtenção dos dados e do gabarito, as duas versões dos arquivos são executadas nas ferramentas e todos os resultados são armazenados. Em seguida, os resultados obtidos são comparados com o gabarito e o número de caracteres de verdadeiros positivos, falsos positivos e falsos negativos é contabilizado. A Tabela 14 e a Tabela 15 apresentam os resultados para as ferramentas IDIFF e WinMerge respectivamente. Nessas tabelas são apresentadas: as operações para cada linha de código, o grão analisado, o trecho de código considerado e a contagem de caracteres.

Tabela 13 – Gabarito do exemplo utilizando a refatoração Inline Method

OPERAÇÃO	CÓDIGO			NC[9]
	Grão	Código Fonte		
		DE	PARA	
DELETED	WORD	moreThanFiveLateDeliveries()	---	28
DELETED	WORD	boolean moreThanFiveLateDeliveries() { return	---	42
MOVED	WORD	_numberOfLateDeliveries > 5	_numberOfLateDeliveries > 5	25
DELETED	WORD	}	---	1
DELETED	WORD	;	---	1
			Total	97

Tabela 14 – Resultados para IDIFF considerando o grão palavra

OPERAÇÃO	CÓDIGO			NC	TP	FP	FN
	Grão	Código Fonte					
		DE	PARA				
DELETED	WORD	moreThanFiveLateDeliveries()	---	28	28	0	0
DELETED	WORD	boolean moreThanFiveLateDeliveries() { return	---	42	42	0	0
MOVED	WORD	_numberOfLateDeliveries > 5	_numberOfLateDeliveries > 5	25	25	0	0
DELETED	WORD	}	---	1	1	0	0
DELETED	WORD	;	---	1	1	0	0
				97	**97**	**0**	**0**

Tabela 15 – Resultados para WinMerge considerando o grão palavra

OPERAÇÃO	Código			NC	TP	FP	FN
	Grão	Código Fonte					
		DE	PARA				
DELETED	LINE	(moreThanFiveLateDeliveries())	---	30	28	2	0
DELETED	LINE	}	---	1	1	0	0
DELETED	LINE	boolean moreThanFiveLateDeliveries() { return _numberOfLateDeliveries > 5;	---	68	42	26	26
ADDED	LINE	---	(_numberOfLateDeliveries > 5)	27	0	27	0
				126	**71**	**55**	**26**

Desta forma o cálculo das variáveis dependentes para o grão palavra possui resultado conforme apresentado na Tabela 16.

[9] NC = Número de Caracteres

Considerando a ferramenta IDIFF: Precisão = $\frac{97}{97+0} = 1$, Cobertura = $\frac{97}{97+0} = 1$ e Aderência = $2 \times \frac{1 \times 1}{1+1} = 1$. Já para ferramenta Winmerge: Precisão = $\frac{71}{71+55} = 0{,}56$, Cobertura = $\frac{71}{71+26} = 0{,}73$ e Aderência = $2 \times \frac{0{,}563492063 \times 0{,}73196}{0{,}563492063 + 0{,}73196} = 0{,}63$.

Tabela 16 – Cálculo de métricas para o grão palavra

	Precisão	Cobertura	Aderência
IDIFF	1	1	1
WinMerge	0,56349	0,73196	0,63677

A Tabela 17 e a Tabela 18 apresentam os resultados obtidos para o exemplo exposto considerando todas as granularidades disponíveis.

Tabela 17 – Resultado de todas as granularidades para IDIFF

	Precisão	Cobertura	Aderência
Grão Linha	48%	73%	58%
Grão Palavra	100%	100%	100%
Grão Caractere	100%	100%	100%

Tabela 18 – Resultado de todas as granularidades para WinMerge

	Precisão	Cobertura	Aderência
Grão Linha	48%	73%	58%
Grão Palavra	56%	73%	64%
Grão Caractere	58%	73%	65%

5.4 **ANÁLISE ESTATÍSTICA**

A análise estatística foi realizada com o intuito de descrever dados e realizar inferências. A principal ideia é comparar resultados das variáveis dependentes (i.e., Precisão, Cobertura e Aderência) nas ferramentas IDIFF e WinMerge após execução dos experimentos em diferentes cenários da refatoração de software.

Todos os testes foram realizados no ambiente de software livre R[10], bastante utilizado para análises estatísticas e construção de gráficos, através da IDE RStudio[11]. A análise foi apoiada por Evandro Lopes, formando do curso de estatística da UFF. Os comandos

[10] http://www.r-project.org/
[11] http://www.rstudio.com/

executados e os resultados dos testes são apresentados com o objetivo de ilustrar a utilização da IDE, apoiando assim outros engenheiros de software que venham a utilizá-la.

5.4.1 **TESTE DE NORMALIDADE**

Existem dois diferentes tipos de erros cometidos na tomada de decisão, durante o teste de hipótese: erro tipo I, quando a hipótese nula é verdadeira porém é rejeitada pelo teste, e erro tipo II, quando é aceita a hipótese nula falsa. Neste ponto é importante ressaltar o conceito do Poder do Teste, que é a probabilidade $1 - \beta$ de detectar uma hipótese H_0 falsa, onde β é a probabilidade de cometer o Erro Tipo II. Diante disso é realizada análise do pressuposto de normalidade dos resultados para decisão entre o uso de testes paramétricos ou não-paramétricos. Com este intuito foi utilizado o teste de Shapiro-Wilk (CONOVER, 1999). As hipóteses para o teste são as seguintes:

$$H_0: A\ amostra\ x_1, x_2, \dots, x_n \qquad possui\ distribuição\ normal$$
$$H_1: A\ amostra\ x_1, x_2, \dots, x_n \qquad não\ possui\ distribuição\ normal$$

No R, o teste é feito através do comando $shapiro.test(x)$, onde x é o vetor contendo o conjunto de dados a serem testados. Conforme pode ser visto na Figura 50.

```
                    Shapiro-Wilk normalitty test

data: x1
w = 0.889, p-value = 6.981e-06
```

Figura 50 – Output do R para a realização do teste de Shapiro-Wilk

Como saída, é fornecido o valor da estatística W[12] do teste de Shapiro-Wilk e seu respectivo *p-value*[13]. Caso este seja menor que o nível de significância α, rejeita-se a hipótese nula, concluindo assim que os dados não possuem distribuição normal.

A suposição de normalidade foi violada para os resultados de cada métrica nas duas ferramentas, pois $p\text{-}value < 0{,}01$, onde o valor apresentado indica que p-value, apesar de ser

[12] A estatística W verifica se a amostra provém de distribuição normal. Menores valores evidenciam normalização dos dados

[13] Menor nível de significância no qual a hipótese nula poderia ser rejeitada para as observações dadas.

maior que zero, é muito pequeno. Como $\alpha = 0{,}05$ e *p-value* $< 0{,}01$ é possível verificar que *p-value* $\leq \alpha$, o que faz com que a hipótese nula seja rejeitada.

Porém um fato importante não observado até o momento refere-se a presença de *outliers*[14] nos dados analisados. Para a análise de normalidade, torna-se necessária a remoção iterativa de *outliers*, tal que, para cada iteração são realizados testes de homocedasticidade (homogeneidade da variância) e normalidade. A análise de homocedasticidade é realizada através do teste F[15], que compara as variâncias das amostras.

Neste contexto H_0= variâncias iguais e H_1= variâncias diferentes, ou seja, p-value ≤ 0.05 indica que as variâncias são diferentes.

Os testes foram realizados para as variáveis Precisão, Cobertura e Aderência, considerando as granularidades linha, palavra e caracter. A Tabela 19 exibe valores do p-value para testes de normalidade após retirada de *outliers*. Na ferramenta WinMerge, considerando a variável cobertura no grão linha, todos os valores ficaram idênticos. A Tabela 20 apresenta valores de p-value para o teste de variância. A título de exemplo, , a Figura 51 exibe o resultado apresentado no R para os testes de variância e normalidade após primeira retirada de *outliers* no grão Palavra, considerando a variável Precisão. Com a análise de resultados, é possível verificar que, apesar de alguns cenários apresentarem variâncias iguais, não foi possível identificar normalidade nas distribuições.

Tabela 19 – Valores de p-value para teste de normalidade

	IDIFF			**WinMerge**		
	Precisão	**Cobertura**	**Aderência**	**Precisão**	**Cobertura**	**Aderência**
Linha	0.025	0.723	0.075	0.004	--	0.001
Palavra	< 0.001	< 0.001	0.01	< 0.001	< 0.001	< 0.001
Caractere	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

Tabela 20 - Valores de p-value para teste de variância

	Precisão	**Cobertura**	**Aderência**
Linha	0.8124	0.6663	0.8902
Palavra	< 0.001	0.7909	0.03625
Caractere	0.2614	0.3023	0.3202

Diante disso, testes não-paramétricos foram utilizados para anaálise estatísticas dos dados. O teste de Wilcoxon[16] foi utilizado para a comparação de médias e o teste de

[14] http://cran.r-project.org/web/packages/outliers/outliers.pdf
[15] http://stat.ethz.ch/R-manual/R-patched/library/stats/html/var.test.html
[16] http://www.r-tutor.com/elementary-statistics/non-parametric-methods/wilcoxon-signed-rank-test

Friedman[17] para a comparação de médias entre grupos. Existem outros testes não paramétricos, porém o teste de Wilcoxon foi escolhido pois os dados são pareados, ou seja, as duas ferramentas rodam sobre o mesmo cenário.

```
[1] "Teste de variâncias iguais"
            F test to compare two variances

      data:  dataframe[, 2] and dataframe[, 3]
      F = 0.4126, num df = 71, denom df = 71, p-value = 0.0002534
      alternative hypothesis: true ratio of variances is not equal to 1
      95 percent confidence interval: 0.2581343 0.6594535
      sample estimates: ratio of variances 0.4125864

[1] "Teste de normalidade para o IDIFF"
            Shapiro-Wilk normality test
      data:  dataframe[, 2]
      W = 0.6468, p-value = 7.062e-12

[1] "Teste de normalidade para o Winmerge"
            Shapiro-Wilk normality test
      data:  dataframe[, 3]
      W = 0.8705, p-value = 2.422e-06
```

Figura 51 – Output do R para a realização do teste F e teste de normalidade

De acordo com Casella e Berger (2001):

- **Teste de *Wilcoxon:*** Comparação pareada que classifica os valores absolutos das diferenças entre os pares e calcula estatística sobre diferenças negativas e positivas. Utilizado em observações dependentes, ou seja, para um mesmo tópico, são obtidos resultados para ambas as ferramentas.

- **Teste de *Friedman:*** Comparações múltiplas que fornece estatísticas descritivas para diferentes variáveis. Normalmente a hipótese nula é de que as variáveis são iguais.

É importante notar que a remoção de *outliers* ocorreu somente para a escolha do teste de hipóteses mais apropriado. *Outliers* podem ser gerados por diversas situações, como, por

[17] http://stat.ethz.ch/R-manual/R-patched/library/stats/html/friedman.test.html

exemplo, dados defeituosos e procedimentos incorretos (BARNETT; LEWIS, 1994). Como não há suspeita que os outliers identificados neste experimento tenham sido decorrentes de medições incorretas ou outras falhas no processo experimental, não há razão para a sua remoção nas fases subsequentes. Desta forma, os *outliers* foram mantidos nas demais fases do processo de análise estatística.

5.4.2 **COMPARAÇÃO DE MÉDIAS**

Esta seção apresenta resultados obtidos pelo teste de *Wilcoxon* para verificar se, em média, os resultados do IDIFF são similares aos do WinMerge. As hipóteses a serem testadas são:

$$H_0: \mu_{IDIFF} = \mu_{WinMerge}$$
$$H_1: \mu_{IDIFF} \neq \mu_{WinMerge}$$

Onde a média é calculada para cada variável dependente em cada parametrização de granularidade. A Tabela 21 apresenta os valores calculados para média e desvio padrão para ambas ferramentas e granularidades.

Tabela 21 – Média e Desvio Padrão por ferramenta e granularidade

	IDIFF			WinMerge		
	LINHA	**PALAVRA**	**CARACTER**	**LINHA**	**PALAVRA**	**CARACTER**
	PRECISÃO					
Média	0,58	0,84	0,75	0,54	0,70	0,74
Desvio Padrão	0,31	0,27	0,34	0,30	0,31	0,30
	COBERTURA					
Média	0,87	0,78	0,58	0,85	0,83	0,79
Desvio Padrão	0,27	0,27	0,32	0,28	0,28	0,29
	ADERÊNCIA					
Média	0,67	0,80	0,63	0,63	0,74	0,74
Desvio Padrão	0,29	0,26	0,32	0,29	0,29	0,28

No caso de rejeição da hipótese nula, é possível afirmar que existe diferença significativa entre as médias. O teste é realizado através do comando $wilcox.test(x, y, paired = T, conf.int = T)$, onde x e y são os vetores dos quais se deseja testar a igualdade entre as médias. A Figura 52 apresenta detalhes do teste realizado.

```
        Wilcoxon signed rank test with continuity correction
data: x1 and x2
V = 574, p-value = 0.002999
Alternative hypothesis: true location shift is not equal to 0
95 percent confidence interval:
-0.2188491  -0.0511941
sample estimates:
(pseudo)median
    -0.1344933
```

Figura 52 – Output do R para a realização do teste de Wilcoxon

Uma vez que se rejeita a hipótese nula, para identificar qual tratamento é superior, podemos olhar para o intervalo de confiança (IC). Se $IC - \alpha\% < 0$, significa que $\mu_{IDIFF} > \mu_{WinMerge}$. Caso contrário $\mu_{WinMerge} > \mu_{IDIFF}$. Caso o *p-value* seja maior que o nível de significância α, a hipótese nula não é rejeitada. Em outras palavras, não temos evidências para afirmar que existe diferença entre os resultados.

Com o intuito de permitir uma visualização sobre os resultados dos testes de hipótese, foram adotados gráficos que exibem a dispersão de cada um dos cenários de refatoração em relação às ferramentas avaliadas. Esses gráficos contêm, no eixo x, o valor da variável dependente em questão para a ferramenta IDIFF, e no eixo y para WinMerge.

Os gráficos gerados a partir do teste de Wilcoxon expostos na Figura 53, Figura 54 e Figura 55, representam a análise da variável precisão. A maior ocorrência de cenários de refatoração abaixo da linha diagonal aponta um possível resultado superior do IDIFF. Caso contrário, indica uma possível superioridade do WinMerge. Essa análise visual é corroborada pela análise estatística, onde o *p-value* é apresentado no canto superior direito de cada figura e a média, apresentada no canto superior esquerdo. A relação entre a numeração apresentada nos gráficos e a refatoração relacionada pode ser encontrada no Apêndice D deste trabalho.

Na Figura 53 e na Figura 54 é possível verificar que o *p-value* é menor que α $(0{,}05)$ e a média do IDIFF é maior, o que indica que o IDIFF apresenta melhores resultados para esta variável nestes cenários.

A Figura 55 mostra um *p-value* alto $(0{,}548)$, o que significa que não é possível rejeitar a H_0 para este cenário. Em outras palavras, indica que não é possível identificar diferença entre as médias analisadas.

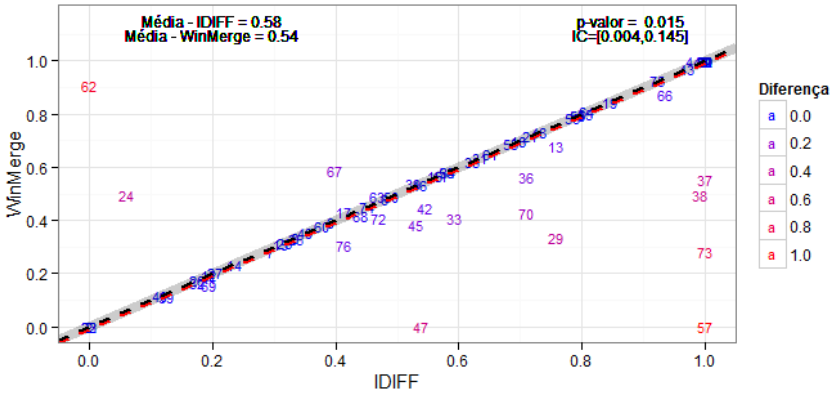


Figura 53 – Teste Wilcoxon – Análise de precisão para o grão linha

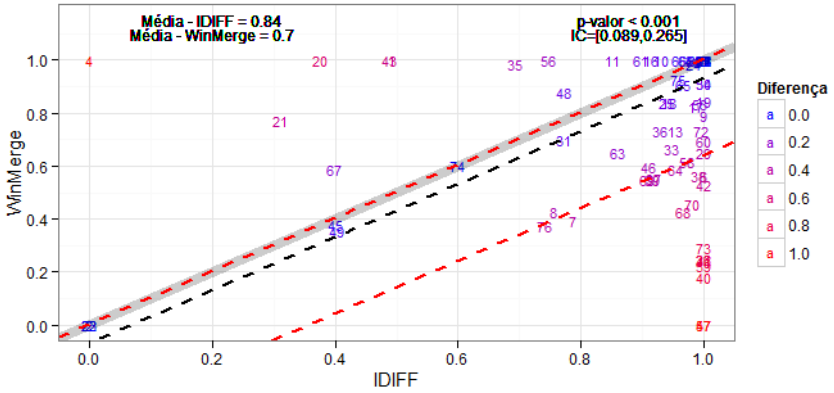


Figura 54 – Teste Wilcoxon – Análise de precisão para o grão palavra

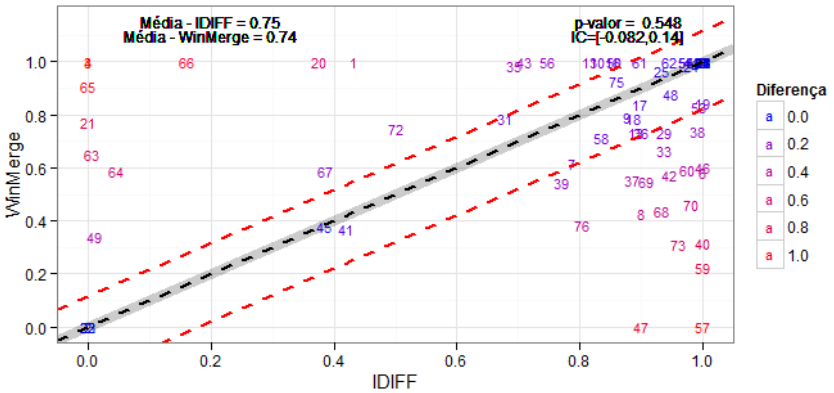


Figura 55 – Teste Wilcoxon – Análise de precisão para o grão caractere

A análise da variável cobertura é realizada conforme Figura 56, Figura 57 e Figura 58. Como pode ser visto no teste pareado para análise de cobertura, o IDIFF não apresenta bons resultados, principalmente ao refinar a granularidade. Uma das causas da redução da cobertura nos resultados é a diminuição da granularidade por etapas, que faz com que algumas semelhanças sejam encontradas de forma incorreta. Também é possível verificar que estatisticamente não há diferença no grão linha, pois $p\text{-}value > \alpha$. Porém, para as demais

granularidades, a média do IDIFF é menor que a média no WinMerge, indicando resultados superiores para o WinMerge na variável cobertura

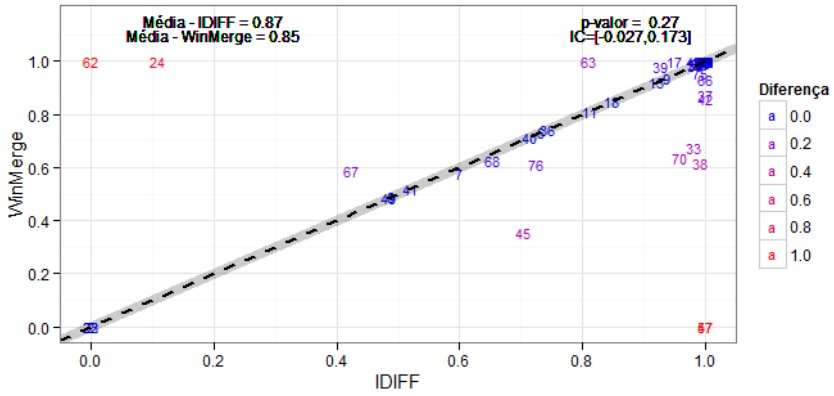


Figura 56 – Teste Wilcoxon – Análise de cobertura para o grão linha

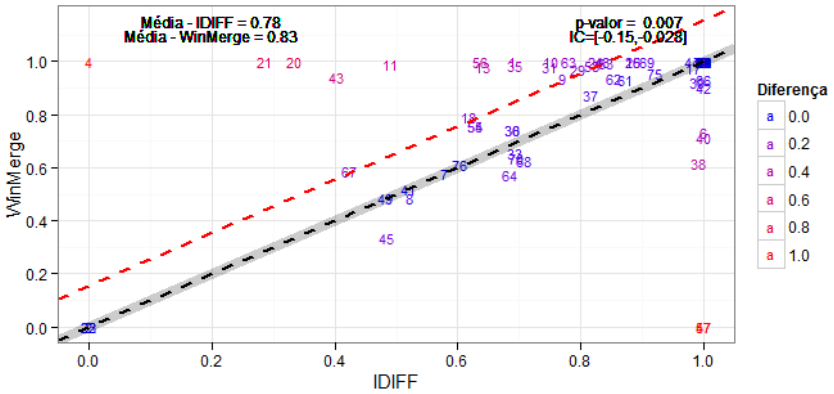


Figura 57 – Teste Wilcoxon – Análise de cobertura para o grão palavra

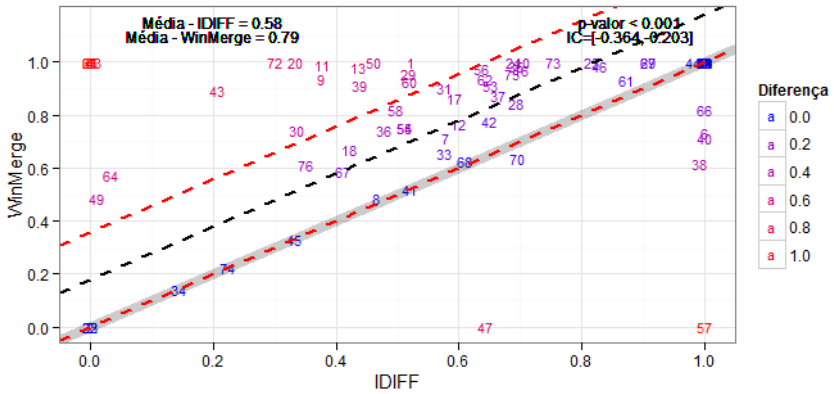


Figura 58 – Teste Wilcoxon – Análise de cobertura para o grão caractere

Finalmente os gráficos dispostos na Figura 59, Figura 60 e Figura 61 representam os resultados para a aderência. Na Figura 59 e na Figura 60 é possível verificar que $p\text{-}value \leq \alpha$

e para ambos os casos a média de IDIFF é maior, indicando melhor resultado para IDIFF. Por sua vez, a Figura 61 indica melhores resultados para o WinMerge, uma vez que sua média é superior e $p\text{-}value < \alpha$.

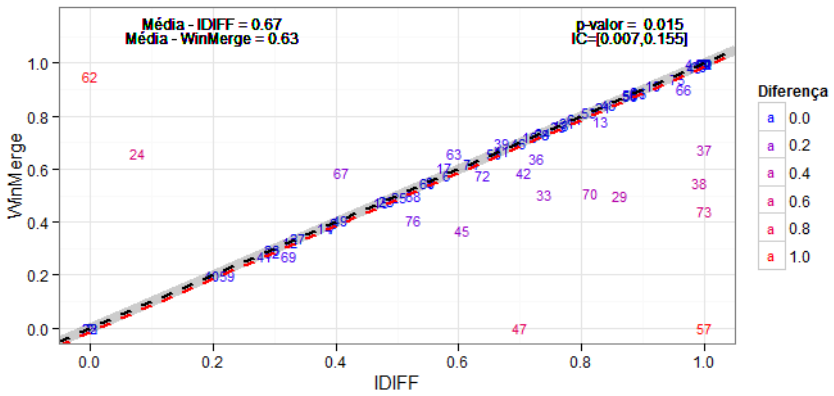


Figura 59 – Teste Wilcoxon – Análise de aderência para o grão linha

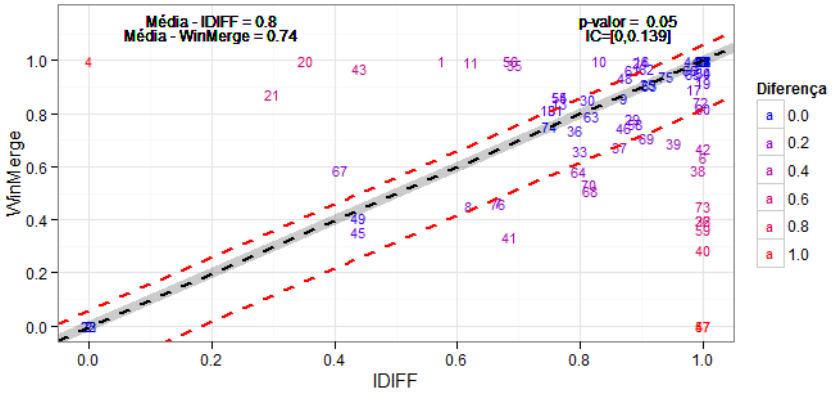


Figura 60 – Teste Wilcoxon – Análise de aderência para o grão palavra

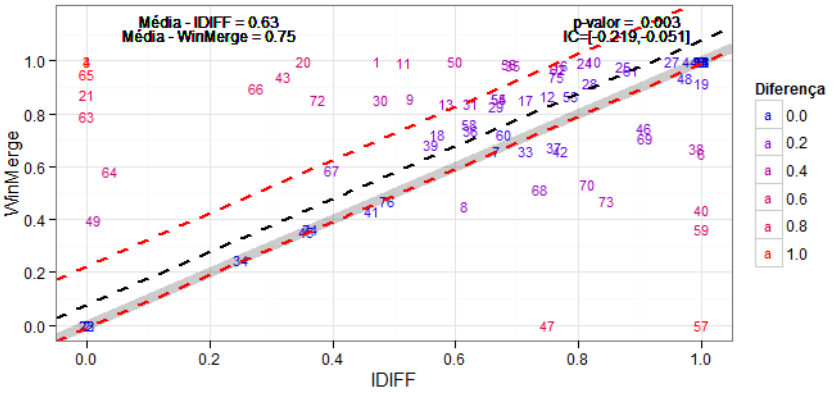


Figura 61 – Teste Wilcoxon – Análise de aderência para o grão caractere

Os gráficos *boxplots* apresentados na Figura 62, Figura 63 e Figura 64 têm como objetivo resumir as distribuições tanto para o IDIFF quanto para o WinMerge, proporcionando outra visualização dos testes descritos anteriormente. Ao verificar os gráficos *boxplot* para precisão, na Figura 62, é possível notar que todas as granularidades apresentam bons resultados para o IDIFF, com destaque para o grão palavra. Outra observação importante a ser feita é referente aos valores discrepantes, conhecidos como *outliers.* É possível notar que o IDIFF apresenta um maior número de *outliers* para todas as variáveis dependentes.

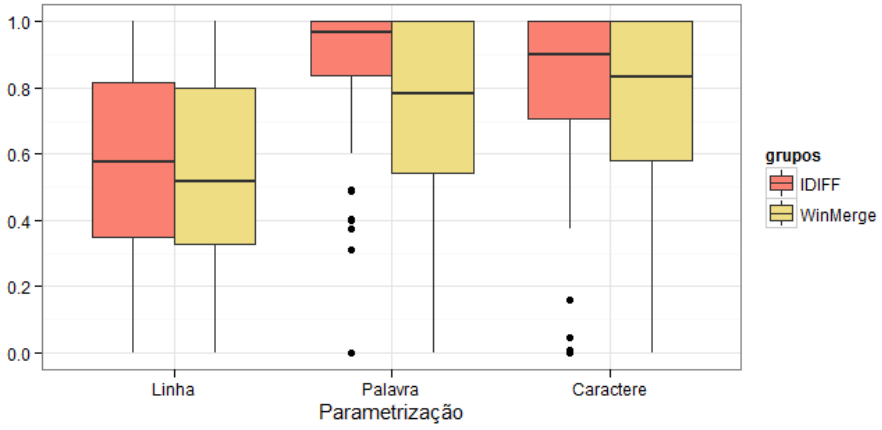


Figura 62 – Boxplot - Análise da variável precisão

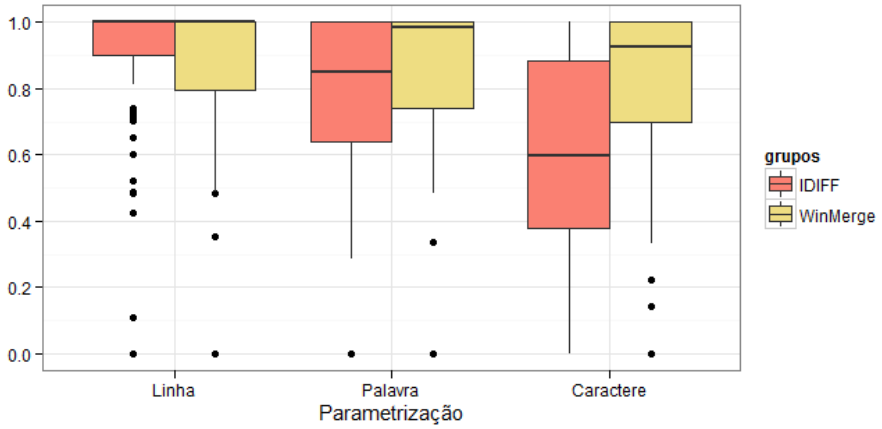


Figura 63 – Boxplot - Análise da variável cobertura

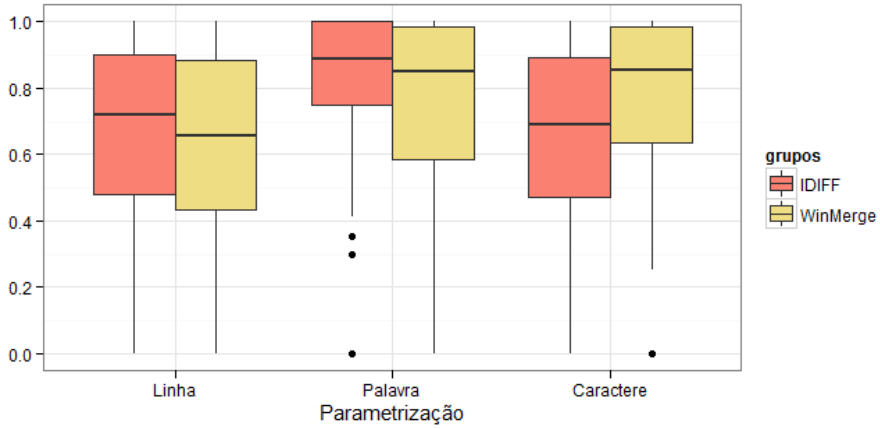


Figura 64 – Boxplot - Análise da variável aderência

Nestes gráficos as caixas representam a parte central da distribuição, que contém 50% dos dados, então quanto maior a caixa, mais dispersos são os dados. A mediana é representada pela linha preta no centro da caixa. Entre a borda superior ou inferior e a mediana estão 25% dos dados. Como, na maior parte dos gráficos, a mediana não está encontra-se no meio, isto indica que as distribuições não são simétricas para os experimentos realizados. Por fim, os *outliers* são dados que se distanciam dos demais, ficando a mais de 1,5 de amplitude interquartil (Q3 – Q1) da borda superior ou inferior da caixa.

5.4.2.1 **ANÁLISE DE *OUTLIERS***

A Tabela 22 exibe as refatorações indicadas como *outliers* para a variável precisão nos diferentes tipos de grãos analisados. Em alguns cenários o IDIFF apresenta dados diferentes do esperado devido a identificação de similaridades que não constam no gabarito definido para os experimentos. Esta situação ocorre devido a redução iterativa de granularidade, realizada pelo IDIFF. Em alguns casos a comparação em granularidades finas provoca a identificação de semelhanças incorretas, principalmente com a utilização de caracteres.

A Tabela 23 apresenta as refatorações indicadas como *outliers* para variável cobertura para os grãos analisados. No cenário da cobertura, para resultados de ambas as ferramentas, o número de falsos negativos detectados influenciam a redução da cobertura calculada, ocorrendo principalmente devido a ferramenta não identificar similaridades que constam no gabarito.

Na maior maior parte destes cenários, na ferramenta IDIFF, as similaridades foram identificadas porém os experimentos consideraram apenas a comparação pareada selecionada pelo algoritmo Húngaro como similaridade ótima global. Já para a ferramenta WinMerge, a identificação de outliers está diretamente relacionada com a falha na detecção de semelhanças entre arquivos, reduzindo assim o número de verdadeiros positivos encontrados.

Tabela 22 – Outliers para variável precisão

	PRECISÃO	
	IDIFF	**WinMerge**
LINHA	Ø	Ø
PALAVRA	Convert Procedural Design to Objects Extract Hierarchy Tease Apart Inheritance Pull Up Field Pull Up Method Push Down Field Push Down Method Extract Class Inline Class Move Method, Split Loop	Ø
CARACTERE	Extract Hierarchy Separate Domain from Presentation Tease Apart Inheritance Pull Up Method Push Down Field Push Down Method Move Method Replace Type Code With Class Replace Type Code With State Strategy Replace Type Code With Subclasses Self Encapsulate Field	Ø

A Tabela 24 apresenta as refatorações indicadas como *outliers* para variável aderência para os grãos analisados durante os experimentos. Estes cenários refletem situações onde a precisão e cobertura diferem significativamente, ou seja alta precisão e baixa cobertura e vice e versa.

Tabela 23 - Outliers para variável cobertura

	COBERTURA	
	IDIFF	**WinMerge**
LINHA	Extract Hierarchy Inline Temp Introduce Explaining Variable Push Down Field Push Down Method Replace Delegation With Inheritance Reverse Conditional Inline Class Move Method Replace Subclass With Fields Split Loop Consolidate Conditional Expression	Extract Hierarchy Push Down Field Push Down Method Inline Class Introduce Local Extensions Reduce Scope Of Variable
PALAVRA	Extract Hierarchy Tease Apart Inheritance Push Down Field Push Down Method	Extract Hierarchy Push Down Field Push Down Method Inline Class Introduce Local Extensions Reduce Scope Of Variable
CARACTERE	Ø	Extract Hierarchy Push Down Field Push Down Method Rename Method Introduce Local Extensions Reduce Scope Of Variable Remove Double Negative

Tabela 24 - Outliers para variável aderência

	ADERÊNCIA	
	IDIFF	**WinMerge**
LINHA	∅	∅
PALAVRA	Extract Hierarchy Tease Apart Inheritance Pull Up Field Pull Up Method Push Down Field Push Down Method	∅
CARACTERE	∅	Extract Hierarchy Push Down Field Push Down Method Introduce Local Extensions Reduce Scope Of Variable

5.4.3 **COMPARAÇÃO DE MÉDIAS EM GRUPOS POR GRANULARIDADE**

O propósito desta subseção é identificar a configuração mais apropriada para cada ferramenta, através da comparação de médias entre grupos. Como verificado anteriormente, os dados não possuem distribuição normal, então é novamente necessário utilizar testes não paramétricos. O teste escolhido para a comparação foi o teste de Friedman e as seguintes hipóteses foram definidas:

$$H_0: \mu_{linha} = \mu_{palavra} = \mu_{caractere}$$
$$H_1: Alguma\ das\ médias\ é\ diferente$$

Em todos os casos o resultado para o teste de Friedman rejeitou a hipótese nula, o que diz que a média é diferente para pelo menos uma das parametrizações. No R, o teste de Friedman pode ser feito com o comando $friedman.test(M)$, tal que M é uma matriz onde cada coluna contém uma parametrização, ou seja, uma coluna para cada tratamento da variável independente. As saídas fornecidas são o valor da estatística de teste chi-quadrado de Friedman e o respectivo *p-value*, conforme pode ser visto na Figura 65.

```
Friedman rank sum test

data: fried.IDIFF
Friedman chi-square = 33.6571, df = 2, p-value = 4.914e-08
```

Figura 65 – Output do R para a realização do teste de Friedman

Como o teste de Friedman identificou diferenças, o passo seguinte consiste em fazer comparações 2 a 2 para identificar quais são essas diferenças. Esta comparação é realizada através da análise numérica das médias calculadas de acordo com as variáveis dependentes e as granularidades expostas, conforme apresentado na Tabela 25.

Tabela 25 – Médias calculadas para cada granularidade e ferramenta

	IDIFF			WinMerge		
	Linha	**Palavra**	**Caractere**	**Linha**	**Palavra**	**Caractere**
Precisão	0,58	0,84	0,75	0,54	0,70	0,74
Cobertura	0,87	0,78	0,58	0,85	0,83	0,79
Aderência	0,67	0,80	0,63	0,63	0,74	0,75

Contudo, ao fazer comparações 2 a 2 existe um maior risco de incidir em Erro Tipo I, pois na medida em que o número de hipóteses de um teste aumenta, a probabilidade de rejeição da hipótese nula, quando a mesma é verdadeira, também aumenta. Para lidar com esse problema, é adotada a correção de Bonferroni (CASELLA; BERGER, 2001), que consiste em modificar o nível de significância de tal forma que ao realizar as comparações múltiplas, o Erro Tipo I não seja multiplicado.

A correção diz que, para um nível de significância α, o valor a ser utilizado em cada comparação é de $\frac{\alpha}{n}$, onde n é o número de testes a ser realizado. Como apresentado anteriormente, neste trabalho foi adotado $\alpha = 0{,}05$. Com isso, o novo $\hat{\alpha}$ é $\frac{0{,}05}{3} = 0{,}017$, já que são necessárias três comparações para identificar todas as diferenças entre as parametrizações (linha x palavra, linha x caractere e palavra x caractere).

A Tabela 26 apresenta os resultados das comparações entre duas granularidades de acordo com as variáveis dependentes analisadas. No primeiro quadro, por exemplo, é possível verificar o cálculo do *p-value* para cada comparação pareada entre granularidades, considerando a ferramenta IDIFF e variável Precisão.

Tabela 26 – Resultado do p-value do teste de comparações múltiplas

IDIFF		WinMerge	
Precisão			
Linha (0.58) - Palavra (0.84)	p-valor < 0.001	Linha (0.54) - Palavra (0.7)	p-valor < 0.001
Linha (0.58) - Caractere (0.75)	p-valor = 0.002	Linha (0.54) - Caractere (0.74)	p-valor < 0.001
Palavra (0.84) - Caractere (0.75)	p-valor = 0.002	Palavra (0.7) - Caractere (0.74)	p-valor = 0.01
Palavra > Caractere > Linha		**Caractere > Palavra > Linha**	
Cobertura			
Linha (0.87) - Palavra (0.78)	p-valor = 0.002	Linha (0.85) - Palavra (0.83)	p-valor = 0.042
Linha (0.87) - Caractere (0.58)	p-valor < 0.001	Linha (0.85) - Caractere (0.79)	p-valor = 0.002
Palavra (0.78) - Caractere (0.58)	p-valor < 0.001	Palavra (0.83) - Caractere (0.79)	p-valor = 0.018
Linha > Palavra > Caractere		**(Linha = Palavra) > (Palavra = Caractere)**	
Aderência			
Linha (0.67) - Palavra (0.8)	p-valor < 0.001	Linha (0.63) - Palavra (0.74)	p-valor < 0.001
Linha (0.67) - Caractere (0.63)	p-valor = 0.487	Linha (0.63) - Caractere (0.75)	p-valor < 0.001
Palavra (0.8) - Caractere (0.63)	p-valor < 0.001	Palavra (0.74) - Caractere (0.75)	p-valor = 0.84
Palavra > (Caractere = Linha)		**(Caractere = Palavra) > Linha**	

Os valores entre parênteses são as médias calculadas para cada granularidade, considerando as variáveis dependentes e as ferramentas. O resultado da comparação é apresentado de forma centralizada na parte inferior de cada quadro. No primeiro quadro consta que $Palavra > Caracter > Linha$. Ou seja, Palavra é a melhor granularidade para a variável precisão na ferramenta IDIFF.

5.4.4 COMPARAÇÃO DE MÉDIAS EM GRUPOS POR TIPO DE REFATORAÇÃO

Esta seção visa identificar qual ferramenta apresenta melhores resultados para os grupos de refatorações. Os dados variam de 0% a 100%, e são calculados através da contagem de melhores resultados para cada grupo. Como exemplo, a Tabela 27 apresenta os resultados em relação a precisão para as duas ferramentas considerando o grupo *Big Refactorings*. Nela é possível verificar que os resultados do WinMerge, para o grão palavra, foram superiores nas refatorações *Convert Procedural Design to Objects* e *Tease Apart Inheritance*. Para as demais refatorações, os resultados são idênticos. Neste caso é dito que o WinMerge é superior em 50% e que IDIFF e WinMerge são iguais nos resultados restantes, representados pelo símbolo (=) nas tabelas seguintes.

A Tabela 28 e Tabela 29 e a Tabela 30 são construídas de forma similar ao exemplo exposto e apresentam resultados para as variáveis precisão, cobertura e aderência, respectivamente.

Tabela 27 – Valores de precisão para refatorações do grupo Big Refactoring

	IDIFF			WinMerge		
	Linha	Palavra	Caractere	Linha	Palavra	Caractere
Convert Procedural Design to Objects	1	0,492063492	0,431192661	1	1	1
Extract Hierarchy	0	0	0	0	0	0
Separate Domain from Presentation	1	1	0	1	1	1
Tease Apart Inheritance	1	0	0	1	1	1

Tabela 28 – Análise numérica de resultados por grupo - Variável precisão

Precisão	Quantidade de Refatorações	Linha			Palavra			Caractere		
		IDiff	WinMerge	=	IDiff	WinMerge	=	IDiff	WinMerge	=
Big Refactoring	4	0,00	0,00	100,00	0,00	50,00	50,00	0,00	75,00	25,00
Composing Methods	9	22,22	0,00	77,78	55,56	22,22	22,22	55,56	11,11	33,33
Dealing with generalization	12	8,33	25,00	66,67	33,33	33,33	33,33	25,00	41,67	33,33
Making method calls simpler	17	47,06	5,88	47,06	76,47	11,76	11,76	52,94	17,65	29,41
Moving Features between objects	8	25,00	25,00	50,00	62,50	37,50	0,00	50,00	50,00	0,00
Organizing data	17	23,53	35,29	41,18	47,06	41,18	11,76	29,41	58,82	11,76
Simplifying conditional expression	9	66,67	11,11	22,22	77,78	0,00	22,22	55,56	22,22	22,22

Tabela 29 – Análise numérica de resultados por grupo - Variável cobertura

Cobertura	Quantidade de Refatorações	Linha			Palavra			Caractere		
		IDiff	WinMerge	=	IDiff	WinMerge	=	IDiff	WinMerge	=
Big Refactoring	4	0,00	0,00	100,00	0,00	50,00	50,00	0,00	75,00	25,00
Composing Methods	9	11,11	0,00	88,89	22,22	44,44	33,33	11,11	77,78	11,11
Dealing with generalization	12	0,00	25,00	75,00	8,33	50,00	41,67	0,00	58,33	41,67
Making method calls simpler	17	23,53	5,88	70,59	29,41	35,29	35,29	11,76	64,71	23,53
Moving Features between objects	8	25,00	12,50	62,50	25,00	50,00	25,00	12,50	62,50	25,00
Organizing data	17	11,76	35,29	52,94	17,65	52,94	29,41	11,76	70,59	17,65
Simplifying conditional expression	9	44,44	11,11	44,44	22,22	44,44	33,33	11,11	66,67	22,22

Tabela 30 – Análise numérica de resultados por grupo - Variável aderência

Aderência	Quantidade de Refatorações	Linha			Palavra			Caractere		
		IDiff	WinMerge	=	IDiff	WinMerge	=	IDiff	WinMerge	=
Big Refactoring	4	0,00	0,00	100,00	0,00	50,00	50,00	0,00	75,00	25,00
Composing Methods	9	22,22	0,00	77,78	44,44	33,33	22,22	33,33	55,56	11,11
Dealing with generalization	12	8,33	25,00	66,67	16,67	50,00	33,33	8,33	58,33	33,33
Making method calls simpler	17	47,06	5,88	47,06	70,59	17,65	11,76	35,29	47,06	17,65
Moving Features between objects	8	25,00	12,50	62,50	25,00	50,00	25,00	12,50	62,50	25,00
Organizing data	17	23,53	35,29	41,18	52,94	35,29	11,76	11,76	76,47	11,76
Simplifying Conditional Expression	9	77,78	0,00	22,22	66,67	11,11	22,22	55,56	22,22	22,22

As células na cor vermelha indicam situações onde o valor destacado é maior do que a soma dos outros dois que estão sendo avaliados, evidenciando assim os resultados mais relevantes, ou seja, todas as células com valores maiores que 50% são escritas nesta cor. Por exemplo, considerando o grupo de refatorações *Big Refactorings*, os resultados idênticos de precisão no grão linha prevalecem em 100% dos casos, e o valor é destacado. Além disso é

possível verificar nas tabelas o número de cenários para cada grupo, ou seja, a quantidade de refatorações pertencentes a cada grupo analisado.

A Tabela 31 apresenta os resultados de forma sumarizada. Cada célula da planilha indica a ferramenta que obteve melhor resultado. Caso o resultado seja idêntico, as duas ferramentas são apresentadas. Além disso, o símbolo asterisco (*) sinaliza os casos onde o resultado semelhante prevalece, como pode ser observado na análise do grupo *Big Refactoring*, que apresenta 100% de semelhança entre os resultados das ferramentas. Por este motivo, a célula é composta pelo conteúdo IDIFF = WinMerge*. Por fim, as células azuis e vermelhas indicam melhores resultados para IDIFF e WinMerge, respectivamente. O símbolo de igualdade (=) indica que as ferramentas apresentaram o mesmo percentual de resultados superiores, já o símbolo asteristico (*) indica os cenários nos quais as ferramentas apresentaram resultados idênticos. Nos casos em que WinMerge supera o IDIFF, foi verificado um grau elevado de igualdade entre os resultados. Por exemplo, na análise do grupo *Dealing with generalization*, o WinMerge exibe resultados melhores em 25%, enquanto o IDIFF possui 8,33% resultados superiores, porém em 66,67% dos casos ambas as ferramentas apresentaram resultados semelhantes. Isto indica que, de modo geral, o IDIFF não reduz significativamente a cobertura e promove uma melhoria na precisão da análise de resultados.

Tabela 31 – Análise de melhores resultados considerando grupos de refatorações.

Grupos		Precisão	Cobertura	Aderência
Linha	Big Refactoring	IDIFF = WinMerge*	IDIFF = WinMerge*	IDIFF = WinMerge*
	Composing Methods	IDIFF*	IDIFF*	IDIFF*
	Dealing with generalization	WinMerge*	WinMerge*	WinMerge*
	Making method calls simpler	IDIFF*	IDIFF*	IDIFF*
	Moving Features between objects	IDIFF = WinMerge*	IDIFF*	IDIFF*
	Organizing data	WinMerge*	WinMerge*	WinMerge*
	Simplifying conditional expression	IDIFF	IDIFF*	IDIFF
Palavra	Big Refactoring	WinMerge*	WinMerge*	WinMerge*
	Composing Methods	IDIFF	WinMerge	IDIFF
	Dealing with generalization	IDIFF = WinMerge*	WinMerge	WinMerge
	Making method calls simpler	IDIFF	WinMerge*	IDIFF
	Moving Features between objects	IDIFF	WinMerge	WinMerge
	Organizing data	IDIFF	WinMerge	IDIFF
	Simplifying conditional expression	IDIFF	WinMerge	IDIFF
Caractere	Big Refactoring	WinMerge	WinMerge	WinMerge
	Composing Methods	IDIFF	WinMerge	WinMerge
	Dealing with generalization	WinMerge	WinMerge	WinMerge
	Making method calls simpler	IDIFF	WinMerge	WinMerge
	Moving Features between objects	IDIFF = WinMerge*	WinMerge	WinMerge
	Organizing data	WinMerge	WinMerge	WinMerge
	Simplifying conditional expression	IDIFF	WinMerge	IDIFF

Na análise da tabela é possível verificar que o WinMerge apresenta bons resultados em grãos mais finos, o que está diretamente relacionado ao fato da análise ser feita em um único grão sem redução iterativa do mesmo, conforme feito no IDIFF. Apesar de apresentar bons resultados para cobertura, esta proposta pode aumentar consideravelmente o número de comparações realizadas pelo algoritmo, quando grandes arquivos são comparados no grão fino. Com a redução da granularidade, o IDIFF possibilita a detecção de diferenças sem a necessidade de um alto poder computacional, porém causa a redução da cobertura dos resultados, uma vez que alguns resultados relevantes não são detectados ou são identificados com semelhanças incorretas.

5.5 **AMEAÇAS A VALIDADE**

Apesar do cuidado em reduzir as ameaças à validade do experimento, há fatores que podem influenciar os resultados. Em relação a validade interna do experimento, a grande quantidade de verificações visuais para calcular as variáveis dependentes, pode ser considerada uma ameaça. Para evitar essa ameaça, as sessões de execução do experimento tiveram tempo limitado para reduzir a influência da fadiga e falta de atenção nos resultados. Existe um risco baixo em relação à confiabilidade das medições, uma vez que a avaliação dos cálculos das variáveis dependentes foi obtida visualmente por comparação com os modelos descritos por Fowler (1999).

No que diz respeito a validade de construção, o uso de diversas refatorações visa reduzir as ameaças à sub-representação de dados. A comparação entre os resultados da ferramenta e os resultados esperados foi realizada de forma objetiva. Todas as execuções foram realizadas com a configuração padrão da ferramenta, exceto pela parametrização da granularidade. Assim, em alguns casos, o IDIFF apresenta desempenho reduzido devido à forma de obtenção dos dados. Por exemplo, é possível citar os casos em que existem semelhanças em duas classes e apenas a melhor escolha, indicada pelo algoritmo húngaro, foi considerada no experimento.

Em relação a validade de conclusão, a escolha de testes estatísticos não-paramétricos para avaliação dos resultados visa reduzir esta ameaça. Finalmente a falta de experiência com projetos de grande porte deixa dúvida se o resultado será satisfatório nesses cenários. Para reduzir a ameaça à validade externa, foram utilizados exemplos do livro base sobre refatorações (FOWLER *et al.*, 1999). Apesar dos exemplos serem reduzidos, as refatorações foram identificadas em cenários reais, o que traz para o experimento situações que provavelmente aconteceriam em um ambiente real de desenvolvimento de software.

5.6 **CONSIDERAÇÕES FINAIS**

Este capítulo apresentou a avaliação do IDIFF, através da análise estatística realizada sobre os valores dos experimentos executados. Os resultados demonstram que o IDIFF fornece resultados mais precisos quando comparado a ferramenta WinMerge, para os cenários expostos pelo Fowler (1999), e possui alguns bons resultados para cobertura e aderência.

Além disso foi possível identificar o grão palavra como melhor variável independente para a ferramenta. Por sua vez o grão caractere, apresentou resultados confusos, indicando semelhanças indesejadas e não relacionadas.

Por fim, foi identificada qual ferramenta apresenta melhores resultados para precisão cobertura e aderência considerando o grupo de refatorações além da granularidade da comparação.

Capítulo 6 – **CONCLUSÃO**

6.1 **CONTRIBUIÇÕES**

Este trabalho apresentou uma nova abordagem para detecção de diferenças em artefatos textuais, independente de linguagem de programação, denominada IDIFF. Essa abordagem faz detecção da similaridade máxima global baseada em conteúdo, por meio do algoritmo Húngaro, permitindo a identificação de mudanças decorrentes de refatorações que transcendem a fronteira do artefato. Por sua vez, a detecção de mudanças em arquivos foi tratada através da utilização do algoritmo ILCS, também proposto no escopo deste trabalho, que permite a identificação de diferenças e movimentações em granularidades distintas. O refinamento automático de granularidades, introduzido pelo ILCS, evita a necessidade de um alto poder computacional para a detecção das diferenças. Por fim, o IDIFF conta também com visualizações especializadas, que possibilitam a utilização de diferentes focos e perspectivas na visualização dos resultados obtidos.

A detecção de similaridades entre arquivos permite a comparação de cada arquivo de um diretório com todos demais arquivos do outro diretório. Através da utilização de algoritmos especializados, a ferramenta proporciona a detecção de similaridades considerando o conteúdo em detrimento do nome. Em situações onde ocorreram refatorações, essa contribuição é de grande relevância, uma vez que possibilita a identificação de mudanças que transcendem as fronteiras dos arquivos. Grande parte das abordagens existentes na literatura considera o nome do arquivo para realização da comparação, impedindo a identificação de mudanças decorrentes de refatorações em diretórios.

A utilização de um algoritmo iterativo no módulo FDiff possibilita a detecção de movimentações e diferenças de forma precisa durante a comparação pareada de artefatos textuais. A precisão da identificação foi avaliada através de um experimento com base em diversas refatorações propostas no livro base de refatoração de software (FOWLER *et al.*, 1999). A ferramenta que implementa a abordagem proposta apresentou, de modo geral, resultados mais precisos que a ferramenta WinMerge, utilizada como *baseline* na comparação.

É importante ressaltar que, durante o experimento, as ações de cada trecho descrito no gabarito foram comparadas de forma exata com os resultados fornecidos pelas ferramentas, sem interpretações de resultados que pudessem favorecer qualquer uma das ferramentas analisadas. A partir disso, é possível indicar que a abordagem IDIFF promove um aumento da

precisão das diferenças detectadas se comparada com abordagens convencionais, como o WinMerge.

Em relação a eficiência, a abordagem IDIFF faz uso iterativo de diferentes granularidades durante a comparação, incluindo linha, palavra e caractere. Essa decisão possibilita a detecção de diferenças com precisão sem a necessidade de um alto poder de processamento. A redução de forma automática visa a eliminação do processamento de grandes blocos idênticos, reduzindo significativamente o tamanho do conteúdo a ser analisado durante a comparação para situações onde há pouca diferença entre as versões. No nível de diretório, a redução é feita por meio da avaliação de arquivos idênticos através da obtenção do SHA1 de cada arquivo. Desta forma, é possível evitar a comparação de arquivos idênticos com os demais arquivos que compõem os diretórios analisados. Para o nível de arquivos, é realizada a redução iterativa das granularidades disponibilizadas (i.e., linha, palavra e caractere). Os grãos finos nunca são executados de forma direta, ou seja, para chegar a identificação no nível de caracteres, todas as linhas similares já foram encontradas, assim como todas as palavras idênticas, reduzindo o conteúdo a ser avaliado no grão mais fino da ferramenta.

Por fim o IDIFF permite análise de resultados em visualizações especializadas. A apresentação dos resultados em diferentes perspectivas permite interpretação das alterações, evitando redundância e retrabalho. O principal objetivo é possibilitar o foco do desenvolvedor na alteração com maior grau de relevância na interpretação os resultados obtidos. Na análise de refatorações, que movem blocos de código de um arquivo para outro, apenas as similaridades apresentam resultados relevantes aos usuários. Por sua vez, a comparação de duas versões de um mesmo arquivo, na maioria dos casos, será melhor analisada com a utilização da perspectiva de diferenças. Conforme visto anteriormente, estudos demonstram que apenas 2% do conteúdo dos arquivos divergem entre duas versões sucessivas (ESTUBLIER, 2000).

6.2 **LIMITAÇÕES**

Os experimentos realizados durante este estudo demonstram que, em alguns casos, a ferramenta IDIFF apresenta resultados irrelevantes, causando assim uma redução na cobertura dos resultados. Isso ocorre em situações onde similaridades não relacionadas são detectadas pela ferramenta, principalmente na utilização da granularidade caractere.

Além disso, a ferramenta IDIFF apresenta alguns efeitos colaterais. Por exemplo, os caracteres especiais “(”, “)”, “{”, “}”, “[” e “]” são apresentados como semelhanças ou diferenças na comparação pareada e múltipla, podendo causar poluição dos resultados obtidos, dificultando assim a compreensão da manutenção realizada. Este problema ocorre principalmente pela ausência do conhecimento da gramática utilizada na linguagem de programação dos arquivos de entrada. Para atenuar este problema, foi desenvolvida a parametrização de separadores, evitando a análise dos mesmos como diferenças e similaridades.

6.3 **TRABALHOS FUTUROS**

Após a construção do IDIFF é possível descrever novos focos de pesquisas a partir da abordagem apresentada. Novos experimentos, modificações internas, melhorias de performance, alteração do algoritmo e inclusão da possibilidade da realização de merge e especialização da ferramenta são as principais propostas para realização de trabalhos futuros.

Em relação aos experimentos, é possível citar a utilização de projetos reais, de forma controlada, para identificação dos resultados esperados. Outra análise interessante diz respeito a construção de um *survey* para identificar o grau de compreensão e satisfação dos usuários na utilização da ferramenta.

No que diz respeito a melhorias e modificações internas, é proposta a análise sintática associada a utilização dos algoritmos propostos nesta abordagem. O conhecimento da gramática da linguagem de programação, utilizada nos arquivos de entrada, possibilita não só a detecção de similaridades como também a interpretação dos resultados obtidos, reduzindo assim a identificação de diferenças irrelevantes.

Além disso, também é levantada a hipótese da utilização de processamento paralelo para aumento na performance dos algoritmos, através da utilização de computadores *multi-core* e placas de vídeo com um considerável número de processadores em GPU.

A especialização da ferramenta, através da possibilidade de receber como entrada a gramática referente a linguagem desejada, permitiria a análise no nível sintático, possibilitando melhor compreensão das diferenças encontradas. Ainda neste cenário, é possível vislumbrar não só a identificação de diferenças em artefatos textuais, mas também em arquivos que seguem formatação específica e *tags* definidas, como, por exemplo, arquivos XML. Com a possibilidade da parametrização da gramática e regras a serem utilizadas, qualquer tipo de arquivo, exceto os binários, poderia ser tratado pela ferramenta.

O IDIFF é baseado no *two-way diff*, que visa somente a comparação entre duas versões não relacionadas. Desta forma é proposto como trabalho futuro a adaptação do IDIFF para utilização de *three-way diff*, ou seja, a utilização de uma versão base como ancestral comum das duas versões que estão sendo comparadas. Esta estratégia a princípio reduziria a detecção de resultados irrelevantes e possibilitaria melhor compreensão das diferenças expostas.

A partir dos experimentos realizados com a ferramenta IDIFF, foi possível identificar que os resultados apresentados são precisos quando comparados a ferramenta WinMerge, escolhida como *baseline*. Como isso, torna-se natural a proposta de criação de uma ferramenta de merge auxiliada pela proposta deste trabalho. A detecção precisa das mudanças permitiria uma maior facilidade na junção de versões e uma melhor compreensão facilitaria a tomada de decisão na resolução de conflitos.

Por fim, a partir do IDIFF poderia ser feito um detector de refatorações, que fizesse uso de expressões regulares para identificar padrões de adição, deleção, modificação e movimentação de trechos de código. Esses padrões poderiam indexar um catálogo de refatorações, indicando se houve refatoração e qual refatoração foi realizada.

Todas as opções indicadas como trabalhos futuros apresentam o mesmo objetivo, que consiste em possibilitar uma compreensão eficaz e eficiente da diferença entre artefatos de software.

REFERÊNCIAS

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. A Differencing Algorithm for Object-Oriented Programs. **Automated software engineering**, ASE, , 2004. p. 2–13.

ARAÚJO, M. A. P.; MONTEIRO, V. F.; TRAVASSOS, G. H. Towards a Model to Support Studies of Software Evolution. **Empirical software engineering and measurement**, ESEM, , 2012. v. 1, p. 281–290.

BAEZA-YATES, R.; RIBEIRO-NETO, B. *Modern Information Retrieval*. 1st. ed. New York, USA: Addison Wesley, 1999.

BARNETT, V.; LEWIS, T. *Outliers in Statistical Data*. 3. ed. USA: Wiley, 1994.

BENNETT, K. H.; RAJLICH, V. T. Software Maintenance and Evolution: A Roadmap. **International conference on software engineering**, ICSE, , 2000. p. 73–87.

BIOLCHINI, J.; MIAN, P. G.; TRAVASSOS, G. H. *Systematic Review in Software Engineering*. Technical Report, nº RT - ES 679 / 05. Rio de Janeiro, RJ: COPPE, UFRJ, 2005.

CASELLA, G.; BERGER, R. L. *Statistical Inference*. 2nd. ed. Thomson Learning: Duxbury Press, 2001.

CONOVER, W. J. *Practical Nonparametric Statistics*. 3rd. ed. New York, NY, USA: Wiley, 1999.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd. ed. The Massachusetts Institute of Technology: The MIT Press, 2009.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. Finding Refactorings Via Change Metrics. **ACM SIGPLAN Notices**, ACM Digital Library, , 2000. v. 35, p. 166–177.

DIG, D.; COMERTOGLU, C.; MARINOV, D.; JOHNSON, R. Automated Detection of Refactorings in Evolving Components. **European conference on object-oriented programming**, ECOOP, , 2006. v. 4067, p. 404–428.

ESTUBLIER, J. Software Configuration Management: A Roadmap. **Proceedings of the Conference on The Future of Software Engineering**, International conference on software engineering. ICSE, , 2000. p. 279–289.

ESTUBLIER, J.; GARCIA, S. Concurrent Engineering Support in Software Engineering. **Automated software engineering**, ASE, , 2006. p. 209 – 220.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. *Refactoring: Improving the Design of Existing Code*. Canada, USA: Addison-Wesley, 1999.

GLASS, R. L.; VESSEY, I.; RAMESH, V. Research in Software Engineering: an Analysis of the Literature. **Information and software technology**, Science Direct, , 2002. v. 44, p. 491–506.

HORWITZ, S. Identifying the Semantic and Textual Differences Between Two Versions of a Program. **Programming language design and implementation**, PLDI, , 1990. p. 234–245.

HUNT, J. W.; MCILROY, M. D. *An Algorithm for Differential File Comparison*. Computing science technical report, n^{o} CSTR 41. Murrray Hill, NJ: Bell Laboratories, 1976.

HUNT, J. W.; SZYMANSKI, T. G. A Fast Algorithm for Computing Longest Common Subsequences. **Communications of the ACM**, ACM, , Maio 1977. v. 20, p. 350–353.

JACKSON, D.; LADD, D. A. Semantic Diff: A Tool for Summarizing the Effects of Modifications. **International conference on software maintenance**, ICSM, , 1994. p. 243–252.

KUHN, H. W. The Hungarian Method for the Assignment Problem. **Naval research logistics quarterly**, 10.1002/nav.3800020109, , 1955. v. 2, p. 83–97.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd. ed. USA: Pearson, 2005.

LEHMAN, M. M. Programs, Life Cycles, and Laws of Software Evolution. **Proceedings of the IEEE**, IEEE, , set. 1980. v. 68, p. 1060–1076.

MALPOHL, G.; HUNT, J. W.; TICHY, W. Renaming Detection. **Automated software engineering**, ASE, , 2003. v. 10, p. 183–202.

MENS, T. A State-of-the-Art Survey on Software Merging. **IEEE transactions on software engineering**, TSE, , 2002. v. 28, p. 449–462.

MENS, T.; GUEHÉNÉUC, Y.; FERNÁNDEZ, J.; D'HONDT, M. Guest Editors' Introduction: Software Evolution. **IEEE Software**, 0740-7459, , 2010. v. 27, p. 22 –25.

MILLER, W.; MYERS, E. W. A file comparison program. **Software: practice and experience**, 1097-024X, , 1985. v. 15, p. 1025–1040.

OBST, W. Delta Technique and String-to-String Correction. *European software engineering conference (ESEC)*. Heidelberg: Springer Berlin, 1987. v. 289. p. 64–68.

PAI, M.; MCCULLOCH, M.; GORMAN, J.; PAI, N.; ENANORIA, W.; KENNEDY, G.; THARYAN, P.; COLFORD JM, J. Systematic Reviews and Meta-Analyses: An Illustrated, Step-by-Step Guide. **The National medical journal of India**, 0970-258X, , dez. 2003. v. 17, p. 86–95.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic Mapping Studies in Software Engineering. **Evaluation and assessment in software engineering**, EASE, , 2008. p. 68–77.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 6th. ed. USA: McGraw-Hill, 2005.

ROCHKIND, M. J. The Source Code Control System. **IEEE transactions on software engineering**, TSE, , 1975. v. 1, p. 364–370.

SOARES, G. Making Program Refactoring Safer. **International conference on software engineering**, ICSE, , 2010. v. 2, p. 521–522.

SPIEGEL, M. *Estatística*. 3. ed. São Paulo, SP: McGraw-Hill, 1994.

TICHY, W. F. RCS: A System for Version Control. *Software - practice and experience*, v. 15, n. 7, p. 637–654, 1985.

TICHY, W. F. The String-to-String Correction Problem With Block Moves. **Transactions on Computer Systems**, TOCS, , 1984. v. 2, p. 309–321.

WEISSGERBER, P.; DIEHL, S. Identifying Refactorings from Source-Code Changes. **Automated software engineering**, ASE, , 2006. p. 231 –240.

WHITE, B. A. *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*. 1st. ed. Boston, USA: Addison-Wesley, 2000.

YANG, W. Identifying Syntactic Differences Between Two Programs. *Software - practice and experience*, v. 21, n. 7, p. 739–755, 1991.

APÊNDICE A – MAPEAMENTO SISTEMÁTICO DA LITERATURA

A.1. INTRODUÇÃO

O termo “mapeamento sistemático” se refere a uma metodologia específica de pesquisa, desenvolvida para coletar e avaliar evidências disponíveis sobre determinado tópico. De forma sucinta, o mapeamento sistemático utiliza métodos explícitos e rigorosos para identificação, avaliação critica e síntese de estudos relevantes sobre um determinado tema (GLASS; VESSEY; RAMESH, 2002). Diante disso, a seguinte questão de pesquisa motivou a construção deste mapeamento:

- Quais são as principais características de ferramentas capazes de realizar *diff* e merge?

Para este trabalho, o processo foi realizado em dupla, reduzindo assim a influência individual na classificação de resultados, além de permitir a construção de resultados não tendenciosos, proporcionando qualidade nas informações obtidas.

A partir disso, os objetivos foram listados e o mapeamento iniciado. O processo é composto pelas fases de planejamento, execução e análise de resultados. Na fase de planejamento é definido um protocolo a ser seguido durante todo o processo, este protocolo descreve um conjunto de passos bem definidos a serem utilizados durante o mapeamento. Com a construção de resultados mais abrangentes e menos dependentes do conhecimento individual, é possível obter uma maior replicabilidade da pesquisa e a atualização das informações obtidas no estudo com o passar do tempo.

Este apêndice se divide em três seções além desta introdução. A seção A.2 apresenta a fase do planejamento e a seção A.3 detalha a fase de execução. Por fim, a seção A.4 apresenta a análise dos resultados.

A.2. FASE DE PLANEJAMENTO

Diante do problema proposto e das questões de pesquisa, o processo de elaboração do protocolo foi iniciado. Este processo envolveu revisões até que um consenso fosse alcançado em relação ao protocolo a ser utilizado.

A estrutura da questão foi elaborada com o intuito de construir uma *string* de busca seguindo o formato PICO (PAI *et al.*, 2003), onde **P** significa população (***P**opulation*), ou seja, pessoas, tipos de projetos e tipos de aplicações afetados pela intervenção; **I** significa

intervenção (***I**ntervention*), levando em consideração a tecnologia de software, ferramentas e procedimentos que irão gerar resultados; **C** significa comparação (***C**omparison*), ou seja, principais alternativas disponíveis. Finalmente, **O** significa a saída (***O**utcome*), e representa o que se deseja observar. Os seguintes dados foram utilizados para a construção da *string* de busca :

> **P =** (Algorithm <OR> Binary Files <OR> Diagram <OR> Directory <OR> Program <OR> Software <OR> Model <OR> Source code <OR> UML <OR> XML)
> **I =** (Change Control <OR> Comparison <OR> Conflict <OR> Delta <OR> Diff <OR> Diff2 <OR> Diff3 <OR> Differencing tool <OR> Merge <OR> Merging <OR> Version <OR> Changes)
> **C = Ø**. Não foram consideradas restrições em tipos de estudos primários para comparação. Este estudo é apenas de caracterização.
> **O = Ø**. O outcome , que consiste nas características dos trabalhos encontrados, não é utilizado na string de busca para não aumentar o número de falsos negativos com a adoção de restrições adicionais. Contudo, essas características são coletadas manualmente durante a leitura e avaliação dos artigos selecionados.

Após a definição das *strings* referentes ao formato PICO, o protocolo utilizado nesta pesquisa foi definido, de forma que ***string de busca* = P AND I**. Ou seja:

***string de busca* =** (Algorithm <OR> Binary Files <OR> Diagram <OR> Directory <OR> Program <OR> Software <OR> Model <OR> Source code <OR> UML <OR> XML) AND (Change Control <OR> Comparison <OR> Conflict <OR> Delta <OR> Diff <OR> Diff2 <OR> Diff3 <OR> Differencing tool <OR> Merge <OR> Merging <OR> Version <OR> Changes)

Considerações no contexto de máquinas de busca influenciaram diretamente a construção da *string* de busca proposta. Desta forma, o protocolo foi adaptado ao formato utilizado na máquina de busca SCOPUS[18]. Segundo o próprio website do SCOPUS, este é o maior banco de dados de resumos e citações de literatura científica e de fontes de qualidade na web .

[18] http://www.scopus.com/

Além do refinamento dos dados da *string*, foram inseridas *tags* específicas para execução do protocolo nesta ferramenta, visando à obtenção de resultados com maior relevância. Com isso, um novo protocolo foi elaborado, segundo a linguagem oferecida pela máquina, conforme apresentado a seguir:

TITLE-ABS-KEY(“Algorithm” OR “Binary Files” OR “Diagram” OR “Directory” OR “Program” OR “Software” OR “Model” OR “Source code” OR “UML” OR “XML”) **AND** (“Change Control” OR “Comparison” OR “Conflict” OR “Delta” OR “Diff” OR “Diff2” OR “Diff3” OR “Differencing tool” OR “Merge” OR “Merging” OR “Version” OR “Changes”))

Este protocolo é avaliado para garantir que o planejamento é viável. Neste trabalho, a definição de artigos de controle influenciou o processo de avaliação do planejamento. Desta forma, a *string* de busca foi definida através de testes de retorno dos artigos de controle, ou seja, caso a expressão de busca retornasse todos os artigos de controle, esta seria considerada apropriada para realização do mapeamento. Neste trabalho foram estabelecidos quatro artigos de controle classificados por ano de publicação, conforme descrito na Tabela 32.

Tabela 32 – Artigos de controle

Título	Autor (es)	Ano
An Algorithm for Differential File Comparison	J. W. Hunt M. D. McIlroy	1976
X-diff an effective change detection algorithm for XML documents	Yuan Wang David J.DeWitt Jin-Yi Cai	2003
A State-of-the-Art Survey on Software Merging	Tom Mens	2002
A three-way merge for XML documents	Tancred Lindholm	2004

O principal objetivo dos artigos de controle é a validação do resultado obtido através da aplicação do protocolo na fonte de busca selecionada, conforme Biolchini et al.(2005). Diversos testes foram conduzidos de forma a garantir que a expressão de busca escolhida estivesse de acordo com objetivos deste estudo. Todo o processo de execução foi realizado no final do ano 2010.

Para a realização desta pesquisa, o idioma inglês foi adotado. A escolha foi realizada, devido à compatibilidade da linguagem com conferências e periódicos internacionais, além da utilização do idioma nas fontes utilizadas para o estudo.

Os critérios de inclusão e exclusão de artigos foram construídos de acordo com a avaliação. Inicialmente, os itens 1, 2 e 3, descritos na Tabela 33, foram os principais critérios de exclusão adotados para a primeira etapa de avaliação.

Tabela 33 – Critérios de exclusão de artigos

Item	Descrição
1	Incompatibilidade ou irrelevância de informações na análise do Resumo e Introdução
2	Incompatibilidade da Área de Conhecimento
3	Incompatibilidade ou irrelevância dos Temas Abordados após leitura completa do artigo

Durante a busca do texto completo dos artigos, foram verificados que alguns deles não estavam disponíveis, de forma gratuita, para leitura. Diante desta situação emails, para os autores, foram enviados no intuito de obter estes documentos, porém parte destas solicitações não foi atendida.

Em relação à inclusão de artigos, um único critério complementar foi estabelecido: a análise das listas de referências bibliográficas dos artigos obtidos com a execução do protocolo. Este critério foi considerado para artigos selecionados após a segunda etapa do mapeamento, e o grau de relevância do assunto abordado foi considerado para adição do artigo.

A partir das informações de publicações selecionadas para o estudo, a estratégia foi a extração de dados relevantes, compatíveis e passíveis de reprodução. Por fim, todos os artigos selecionados nas etapas de avaliação estão disponíveis nos materiais complementares disponibilizados juntamente com este trabalho.

A.3. ETAPAS DE EXECUÇÃO

O processo de seleção de estudos foi dividido em etapas, de forma que cada uma delas considera de maneira distinta os critérios definidos para este mapeamento.

1ª Etapa - seleção dos estudos coletados: enumeração preliminar das publicações, realizada a partir da aplicação da expressão de busca na fonte principal escolhida.

2ª Etapa - seleção dos estudos relevantes (1º filtro): após a identificação das publicações, os resumos (*abstracts*) e as introduções foram lidas e analisadas.

3ª Etapa - seleção dos estudos relevantes (2º filtro): apesar de limitar o universo de busca, o filtro empregado na 2ª etapa não garante que o material coletado seja útil no contexto

da pesquisa. Por isso, as publicações selecionadas na 2ª etapa são lidas por completo, verificando se, de fato, atendem aos critérios definidos.

Na 1ª etapa, após a execução do protocolo na fonte principal de pesquisa foram retornados 513 artigos no total. A leitura e a avaliação foram realizadas em conjunto com o aluno Eraldo Borel, formando da Universidade Federal Fluminense.

Na 2ª etapa foram avaliados, através da leitura do resumo e da introdução, a pertinência ao tema do mapeamento, os temas abordados, as principais características e possíveis críticas referentes ao estudo analisado. A Figura 66 demonstra o percentual de artigos eliminados nesta etapa da avaliação, onde apenas 16% do total de artigos selecionados na primeira etapa foram selecionados na segunda etapa Ou seja, 84 artigos foram filtrados na segunda etapa, tendo 359 foram eliminados por leitura do resumo e 71 por leitura da introdução.

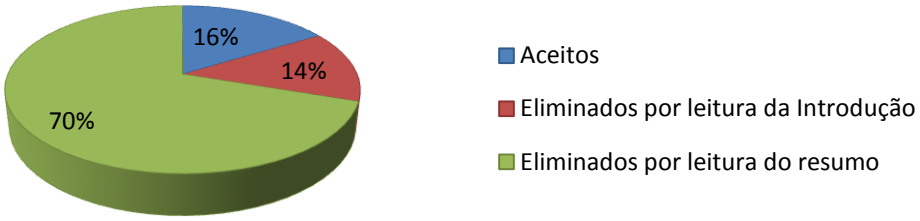


Figura 66 – Avaliação de artigos selecionados na 2ª etapa

Na sequência, os documentos foram lidos, considerando critérios mais rigorosos e de maior grau de relevância. A partir disso, 84 artigos foram processados, na 3ª etapa, conforme a Figura 67. Após leitura e análise realizada na segunda etapa de avaliação, cada publicação aprovada foi examinada individualmente, através da leitura completa dos artigos, porém alguns não estavam disponíveis para leitura ou foram rejeitados por incompatibilidade em relação ao assunto abordado.

Nesta terceira etapa é esperado que os dados da publicação sejam armazenados, de forma sintética. A principal finalidade é expor, de maneira sucinta, dados e características relevantes da publicação, facilitando assim a posterior utilização das informações obtidas durante o processo de mapeamento.

Alguns elementos básicos foram informados para uma melhor identificação da leitura realizada, como, por exemplo, título, autores, data da publicação, referência completa e resumo da publicação.

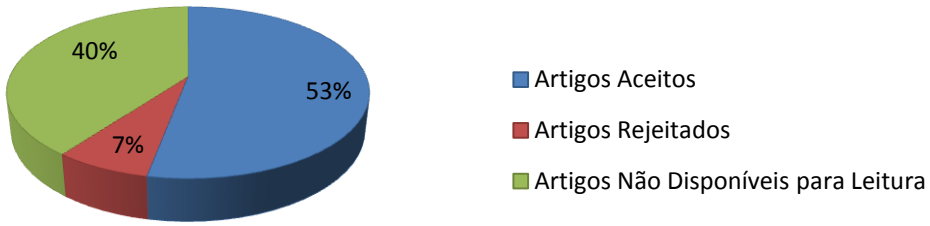


Figura 67 – Classificação de artigos na 3ª etapa

A.4. ANÁLISE DE RESULTADOS

Os resultados obtidos durante o processo de mapeamento possibilitou a listagem de características citadas nas publicações avaliadas conforme pode ser visto na Tabela 34. Esta tabela exibe não só as características como também o número de ocorrências nos artigos aceitos.

Tabela 34 – Listagem de características organizadas pela ocorrência.

Característica	Número de ocorrências
Realizar diff estrutural de código.	8
Detectar movimentação entre linhas de código.	6
Estar disponível como plugin para interfaces de desenvolvimento.	5
Ter boa performance.	3
Indicar diferenças de modo visual.	3
Permitir acesso via linha de comando.	2
Permitir 3-way merge.	2
Realizar diff de modelos UML.	2
Utilizar o GNU DIFF.	2

Estas características respondem a questão de pesquisa: “Quais são as principais características de ferramentas capazes de realizar *diff* e merge?”. Dentre todas as listadas, algumas características foram selecionadas para atendimento neste trabalho. Sendo elas:

- Realizar diff estrutural de código.
- Detectar movimentação entre linhas de código.
- Ter boa performance.
- Indicar diferenças de modo visual.
- Possibilidade de parametrização de dados a serem analisados.

APÊNDICE B – SURVEY

B.1. INTRODUÇÃO

Este apêndice descreve a construção, a distribuição e a análise de um survey sobre ferramentas de *diff* e merge. O principal objetivo é a coleta de opiniões sobre características de ferramentas destinadas à comparação de artefatos no âmbito acadêmico e industrial.

O mapeamento sistemático, descrito no Apêndice A, foca em artigos científicos sobre algoritmos para detecção de diferenças. Complementarmente, para uma análise do estado da prática, este *survey* concentra no que está sendo usado no dia a dia pelos desenvolvedores.

B.2. CONSTRUÇÃO E APLICAÇÃO

O formulário, apresentado no Apêndice C, é composto por questões referentes à experiência e ao uso de diversas ferramentas de comparação. Além disso, possibilita a indicação de características não enumeradas através de campos de texto de livre preenchimento. O *survey* foi construído e disponibilizado no Google Drive[19] e a solicitação de colaboração foi realizada para um público alvo de 80 pessoas e esteve disponível no prazo de 15 dias. Dentre os participantes estavam alunos da Universidade Federal Fluminense estudantes de Ciência da Computação, graduandos e Pós-graduandos, além de pessoas do mercados de trabalho, de empresas públicas e privadas.

B.3. ANÁLISES DAS RESPOSTAS OBTIDAS

No momento da avaliação dos resultados, foram recebidas 63 respostas de usuários com diferentes perfis. A partir disso, foi iniciada a geração de gráficos e tabelas para exibição, análise e interpretação das respostas. Os gráficos e tabelas exibidos a seguir refletem as respostas dos usuários para o formulário disponibilizado, permitindo assim a caracterização dos participantes da pesquisa.

Inicialmente a Figura 68 e a Figura 69 representam as respostas referentes a definição de perfil do usuário participante do *survey*. A identificação da faixa etária e grau de escolaridade têm o objetivo de refinar respostas de acordo com este perfil.

[19] https://drive.google.com/

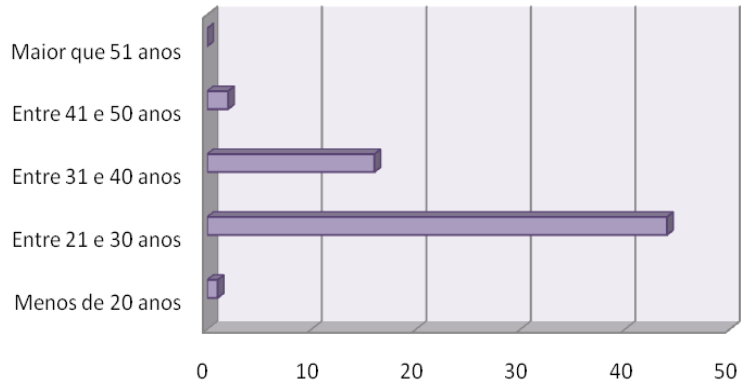


Figura 68 – Análise de faixa etária

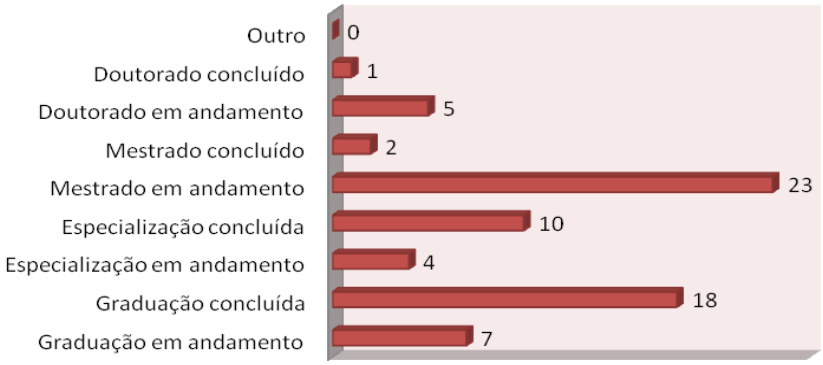


Figura 69 – Grau de escolaridade

Os próximo gráficos apresentam as respostas referentes a questões sobre a utilização da ferramenta. A Figura 70, Figura 71 e Figura 72 indicam a análise do tempo de experiência de acordo com o tipo de projeto: acadêmico ou industrial. Como é possível verificar na Figura 70, foram excluídos da análise os participantes que declararam não possuir experiência neste tipo de ferramentas

Figura 70 – Análise de experiência na utilização de ferramentas de diff e merge

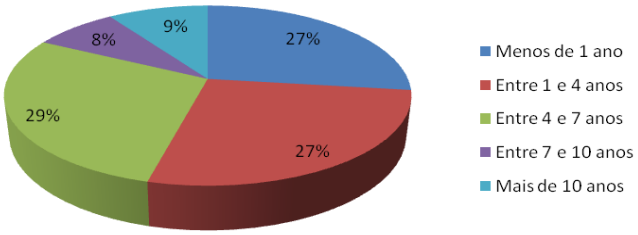


Figura 71 – Análise de tempo de experiência em projetos de indústria

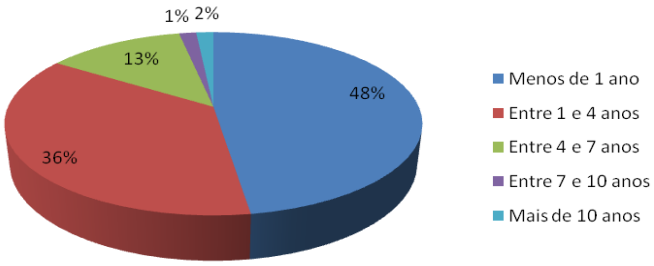


Figura 72 – Análise de tempo de experiência em projetos acadêmicos

A Figura 73 apresenta as respostas em relação ao grau de satisfação na utilização de algumas das principais ferramentas existentes. Neste gráfico é possível observar o alto grau de satisfação na utilização da ferramenta WinMerge, onde aproximadamente 51% dos participantes declaram ter utilizado a ferramenta. Dentre estes, 87,5% indicam boa ou excelente impressão na utilização da mesma. Este foi um dos principais motivadores para escolha da ferramenta como *baseline* para os experimentos.

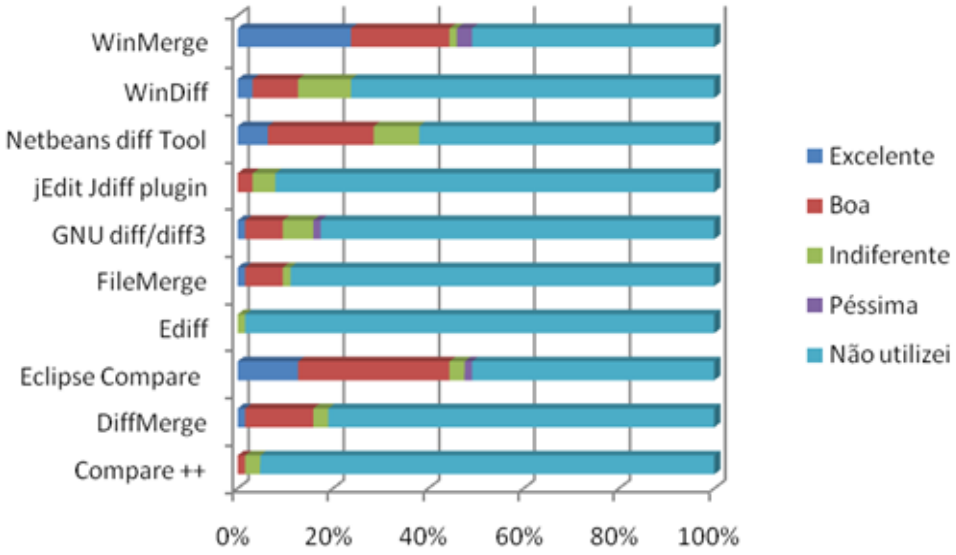


Figura 73 – Análise de satisfação ao utilizar ferramentas de diff e merge

A Figura 74 apresenta os artefatos considerados mais importantes a serem comparados. Dentre eles, se destacam diretórios, que demandam tratamento não-sequencial, e Arquivos, que demandam tratamento sequencial, ambos tratados neste trabalho.

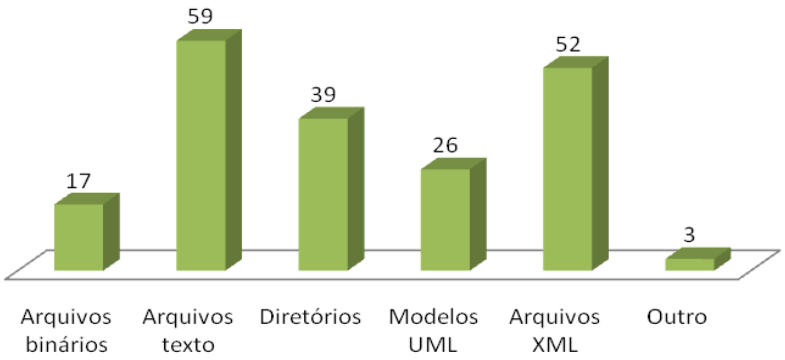


Figura 74 – Análise de quais artefatos são considerados importantes

A Figura 75 e a Figura 76 indicam as características consideradas fundamentais e desejáveis para ferramentas de detecção de diferenças, respectivamente. Na proposta deste trabalho apenas a comparação $three-way$, que considera um ancestral comum, não foi abordada. Todas as demais características foram desenvolvidas.

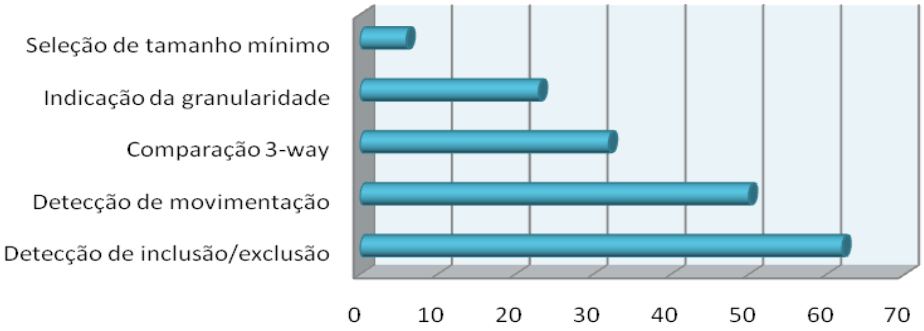


Figura 75 – Análise de características fundamentais a ferramentas de diff e merge

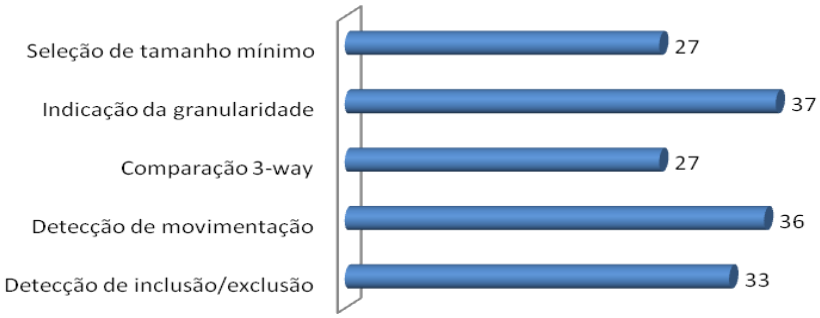


Figura 76 – Análise de características desejáveis

A Tabela 35 e a Tabela 36 enumeram algumas características fundamentais e desejáveis descritas pelos participantes do *survey*. As características atendidas neste trabalho estão destacadas, em negrito e na cor amarela, nestas tabelas. As demais não foram consideradas, pois a proposta não é restrita a linguagens de programação.

Tabela 35 – Características fundamentais não citadas

Indique características não citadas anteriormente
Determinar o que não comparar (por exemplo, espaços em branco / tabs).
Apoio especializado a determinados tipos de arquivos (ex., documentos do word/excel).
Apoio especializado a determinadas linguagens.
Possibilitar comparação de diretórios, mostrando arquivos criados, removidos e alterados.
Comparação entre dois repositórios com mais de um ancestral comum.

Tabela 36 – Características desejáveis não citadas

Indique características não citadas anteriormente
Identificar múltiplas alterações, reconhecendo trechos internos inalterados.
Poder configurar o fato de ignorar alteração em certas linhas (p.ex. arquivos de configuração em linhas com conteúdo que é especifico pela estação de trabalho).
Clareza nos relatórios comparativos. A maioria das ferramentas *diff*/merge, em projeto da faculdade, gerava relatórios de difícil entendimento, ou de visualização difícil dos resultados.
Clareza de interface, sem ambiguidades.
Comparação entre dois repositórios com mais de um ancestral comum.

APÊNDICE C – FORMULÁRIO SURVEY

C.1.1. DESCRIÇÃO DO FORMULÁRIO

Este questionário tem por objetivo coletar opiniões sobre características fundamentais e desejáveis em ferramentas destinadas à comparação de artefatos. Tempo estimado para preenchimento: 3 minutos

---

*Obrigatório

Perfil do Usuário

Nome
Indique seu nome, caso queira se identificar

Idade
Maior que 51 anos

Grau de Escolaridade
* Nível de formação acadêmica

- ☐ Graduação em andamento
- ☐ Graduação concluída
- ☐ Especialização em andamento
- ☐ Especialização concluída
- ☐ Mestrado em andamento
- ☐ Mestrado concluído
- ☐ Doutorado em andamento
- ☐ Doutorado concluído
- ☐ Outro:

Utilização de Ferramentas

Possui experiência referente à utilização de ferramentas de *diff* e merge? *

☐ Sim, em projetos de indústria

☐ Sim, em projetos acadêmicos

☐ Não, não possuo experiência em ferramentas de *diff* e merge

Indique o tempo de experiência em projetos de indústria
Se possuir experiência, indique aqui o período dedicado aos projetos

Indique o tempo de experiência em projetos acadêmicos
Se possuir experiência, indique aqui o período dedicado aos projetos

Classifique a impressão que você obteve ao utilizar as seguintes ferramentas de *diff* e merge *

	Não utilizei	Péssima	Indiferente	Boa	Excelente
Compare++	○	○	○	○	○
DiffMerge	○	○	○	○	○
Eclipse Compare	○	○	○	○	○
Ediff	○	○	○	○	○
FileMerge	○	○	○	○	○
GNU *diff*/*diff3*	○	○	○	○	○
jEdit JDiff plugin	○	○	○	○	○
Netbeans Diff Tool	○	○	○	○	○
WinDiff	○	○	○	○	○
WinMerge	○	○	○	○	○
Tortoise diff	○	○	○	○	○

Outras ferramentas
Indique ferramentas já utilizadas que não constam na questão anterior

Em quais artefatos você considera importante a utilização de ferramentas de *diff* e merge? *

- ☐ Arquivos binários
- ☐ Arquivos texto
- ☐ Diretórios
- ☐ Modelos UML
- ☐ Arquivos XML
- ☐ Outro:

Em sua opinião, quais são as características fundamentais a ferramentas de *diff* e merge?
* Selecione no máximo 3 características

- ☐ Detecção de inclusão/exclusão
- ☐ Detecção de movimentação
- ☐ Comparação three-way (considerando ancestral comum)
- ☐ Indicação da granularidade de execução (letra, palavra, linha)
- ☐ Seleção de tamanho mínimo a ser considerado numa alteração

Indique características não citadas anteriormente
Indique características fundamentais, não citadas.
Se possível indique motivo da sua sugestão

Em sua opinião, quais são as características desejáveis a ferramentas de *diff* e merge?
* Selecione no máximo 3 características

- [ ] Detecção de inclusão/exclusão
- [ ] Detecção de movimentação
- [ ] Comparação three-way (considerando ancestral comum)
- [ ] Indicação da granularidade de execução (letra, palavra, linha)
- [ ] Seleção de tamanho mínimo a ser considerado numa alteração

Indique características não citadas anteriormente
Indique características desejáveis, não citadas.
Se possível indique motivo da sua sugestão

o

Submit

APÊNDICE D – LISTA DE REFATORAÇÕES

E.1.1. REFATORAÇÕES ANALISADAS

Listagem de refatorações analisadas durante os experimentos para cálculo de Precisão, Cobertura e Aderência. A numeração aqui apresentada é utilizada nos gráficos gerados pelos experimentos.

Big Refactoring

1 - Convert Procedural Design to Objects

2 - Extract Hierarchy

3 - Separate Domain from Presentation

4 - Tease Apart Inheritance

Composing Methods

5 - Extract Method

6 - Inline Method

7 - Inline Temp

8 - Introduce Explaining Variable

9 - Remove Assignments to Parameters

10 - Replace Method with Method Object

11 - Replace Temp with Query

12 - Split Temporary Variable

13 - Substitute Algorithm

Dealing with Generalization

14 - Collapse Hierarchy

15 - Extract Interface

16 - Extract Subclass

17 - Extract Superclass

18 - Form Template Method

19 - Pull Up Constructor Body

20 - Pull Up Field

21 - Pull Up Method

22 - Push Down Field

23 - Push Down Method

24 - Replace Delegation With Inheritance

25 - Replace Inheritance With Delegation

Making Method Calls Simpler

26 - Add Parameter

27 - Encapsulate Downcast

28 - Hide Method

29 - Introduce Parameter Object

30 - Parameterize Method

31 - Preserve Whole Object

32 - Remove Parameter

33 - Remove Setting Method

34 - Rename Method

35 - Replace Constructor with Factory Method

36 - Replace Error Code with Exception

37 - Replace Exception with Test

38 - Replace Parameter with Explicit Methods

39 - Replace Parameter with Method

40 - Replace Static Variable with Parameter

41 - Reverse Conditional

42 - Separate Query from Modifier

Moving Features Between Objects

43 - Extract Class

44 - Hide Delegate

45 - Inline Class

46 - Introduce Foreign Method

47 - Introduce Local Extensions

48 - Move Fields

49 - Move Method

50 - Remove Middle Man

Organizing Data

51 - Change Bidirectional Association to Unidirectional

52 - Change Unidirectional Association to Bidirectional

53 - Change Value To Reference

54 - Duplicate Observed Data

55 - Encapsulate Collection

56 - Encapsulate Field

57 - Reduce Scope Of Variable

58 - Replace Array With Object

59 - Replace Assignment With Initialization

60 - Replace Data Value With Object

61 - Replace Magic Number With Symbolic Constant

62 - Replace Subclass With Fields

63 - Replace Type Code With Class

64 - Replace Type Code With State Strategy

65 - Replace Type Code With Subclasses

66 - Self Encapsulate Field

67 - Split Loop

Simplifying Conditional Expressions

68 - Consolidate Conditional Expression

69 - Consolidate Duplicate Conditional Fragments

70 - Decompose Conditional

71 - Introduce Assertion

72 - Introduce Null Object

73 - Remove Control Flag

74 - Remove Double Negative

75 - Replace Conditional With Polymorphism

76 - Replace Nested Conditional With Guard Clauses

E.2. REFATORAÇÕES NÃO ANALISADAS

As seguintes refatorações não foram analisadas, durante os experimentos, devido a falta de dados necessários para construção dos resultados relevantes para análise de Precisão, Cobertura e Aderência.

1. Change Reference to Value
2. Convert Dynamic to Static Construction
3. Convert Static to Dynamic Construction
4. Eliminate Inter-Entity Bean Communication
5. Extract Package
6. Hide presentation tier-specific details from the business tier
7. Introduce a Controller
8. Introduce Business Delegate
9. Introduce Local Extension
10. Introduce Synchronizer Token
11. Localize Disparate Logic
12. Merge Session Beans
13. Move Business Logic to Session
14. Move Class
15. Refactor Architecture
16. Replace Conditional with Visitor
17. Replace Iteration with Recursion
18. Replace Record with Data Class
19. Remove Control Flag
20. Separate Data Access Code
21. Use a Connection Pool
22. Wrap entities with session