UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

JULIANA IORA

PADRÕES DE CORREÇÃO DE DEFEITOS: UM ESTUDO EXPLORATÓRIO

TRABALHO DE CONCLUSÃO DE CURSO

CAMPO MOURÃO – PR
2015

JULIANA IORA

PADRÕES DE CORREÇÃO DE DEFEITOS: UM ESTUDO EXPLORATÓRIO

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Tecnólogo em Sistemas para Internet.

Orientador: Prof. Dr. Reginaldo Ré

CAMPO MOURÃO – PR

2015

DEDICATÓRIA

Dedico esta monografia à minha família pela fé e confiança demonstradas.

Aos meus amigos pelo apoio incondicional.

Aos professores pelo fato de sempre estarem dispostos a ensinar.

Ao orientador pela paciência demonstrada no decorrer do trabalho.

Enfim, a todos que de alguma forma tornaram este caminho mais fácil de ser percorrido.

AGRADECIMENTOS

Quero agradecer, em primeiro lugar, à Deus, pela força e coragem durante toda esta longa caminhada por ter me dado saúde e determinação para superar as muitas dificuldades.

Agradeço também a todos os professores que me acompanharam durante a graduação, em especial ao Prof. Dr. Reginaldo Ré, por toda a paciência e confiança em mim depositadas, ingredientes esses que tornaram possível a realização e conclusão deste trabalho.

Aos meus pais Adriana Iora e Vilmar Iora, e a minha irmã Giovana Iora que nunca mediram esforços para que eu pudesse alcançar este degrau de minha vida, pelo amor, incentivo e apoio incondicional.

Por fim, a todos que direta ou indiretamente fizeram parte da minha formação e conquista, o meu muito obrigado.

RESUMO

IORA, Juliana. PADRÕES DE CORREÇÃO DE DEFEITOS: UM ESTUDO EXPLORATÓRIO. 25 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Sistemas para Internet, Universidade Tecnológica Federal do Paraná. Campo Mourão - PR, 2015.

Este trabalho apresenta um estudo exploratório que tem por objetivo investigar a possibilidade de descrição da correção de *bugs* por meio da aplicação de um ou mais operadores de mutação. Mediante uma análise manual dos arquivos de *log* do jEdit, os quais foram obtidos por meio do sistema de controle do versão do projeto, selecionou-se algumas revisões, as quais puderam ser analisadas recorrendo à aplicação de operadores de mutação e dos padrões de correção descritos por Pan et al. (2008). Para cada revisão obteve-se a porcentagem de cobertura tanto por operadores de mutação como por padrões de correção, sendo que estes últimos apresentaram uma porcentagem média de cobertura de 64% em relação a 36% dos operadores de mutação. Os resultados mostraram que foi realmente possível descrever os *bugs* de um projeto de software pelo uso de operadores de mutação, ou seja, a abordagem investigada no trabalho mostrou-se útil, sendo importante ressaltar a validade da mesma já que os resultados foram obtidos para defeitos reais.

Palavras-chave: Bugs de software, JEdit, Operadores de mutação, Padrões de correção de bugs.

ABSTRACT

IORA, Juliana. BUG FIX PATTERNS: AN EXPLORATORY STUDY. 25 f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Sistemas para Internet, Universidade Tecnológica Federal do Paraná. Campo Mourão - PR, 2015.

This paper presents an exploratory study in order to investigate the possibility to describe bug fix by using a mutant operator or a set of mutant operators. Through a manual analysis of the jEdit log files, which were obtained from the version control system, it was selected some revisions that were evaluated by employing mutant operators and the bug fix patterns proposed by Pan et al. (2008). For each revision we evaluated if it could be covered by mutation operators and by bug fix patterns. According to the analysis around 36% of the revisions were covered by mutation operators while about 64% could be covered by the bug fix patterns. The data suggest that it was possible to fix bugs from a project by mutation operators, that is, the approach explored may be useful since the dada were obtained from real bugs which were extracted from the project.

Keywords: Bug fix patterns, JEdit, Mutation operators, Software bugs.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
SVN	Apache Subversion
HTML	Hyper Text Markup Language
XML	Extensible Markup Language
CSV	Comma Separated Values
HAZOP	Hazard and Operability Studies

SUMÁRIO

1. INTRODUÇÃO

Muito progresso tem sido feito em teste de software ao longo dos últimos anos (DEBROY e WONG, 2010; JUST, KAPFTHAMMER e SCHWEIGGERT, 2012; DELAMARO, OFFUTT e AMMANN, 2014). No entanto, mesmo com as propostas para produzir software com qualidade, ele não está livre de conter defeitos, que causam falhas, travamentos ou resultados incorretos, comprometendo não só a confiabilidade, mas também a segurança do software (DEBROY e WONG, 2014, p. 45). Além do esforço de teste, o qual visa encontrar defeitos, conforme descrito por Liu et al. (2013), há o esforço de correção de um *bug*[1],que constitui um processo desafiador e demorado.

Alguns estudos têm sido realizados no sentido de aperfeiçoar o processo de correção de *bugs*, como é o caso de Debroy e Wong (2014), que propuseram uma estratégia para produzir potenciais correções para programas defeituosos através da combinação das ideias de mutação e localização de falhas. Em outro trabalho, Kim et al. (2006), por meio da inspeção do histórico de alterações do software, extraíram e armazenaram padrões observados tanto nos códigos fonte defeituosos como nos tipos de modificações realizadas para corrigir esses códigos, a fim de construir um banco de dados que pudesse ser utilizado para encontrar novos *bugs* e sugerir correções para os mesmos.

Pan et al. (2008), por sua vez, exploraram os padrões de correção de defeitos a partir da análise do histórico de alterações de projetos de código aberto. Essa análise envolveu a inspeção dos pares de *bug hunks* e seus correspondentes pares de *fix hunks* das revisões. Um *bug hunk* indica a seção do código na versão defeituosa que é modificada ou excluída, e seu correspondente *fix hunk* indica o código na versão de correção que corrige o *bug*.

Muito embora sejam trabalhos importantes, os padrões de correção de defeitos propostos por Pan et al. (2008) e Kim et al. (2006), que utilizam a mineração dos dados do histórico de alterações do software, não descrevem completamente a correção de defeitos. Pan et al. (2008) constataram que 45,7% a 63,6%, apenas, das correções

[1]O termo *bug* refere-se, na prática, a algo que está implementado de maneira incorreta no código fonte. Embora engano, defeito, erro, e falha evoquem ideias parecidas, são usados para designar conceitos distintos (IEEE, 1990). Neste trabalho, os termos engano, defeito, e erro serão usados como bug (causa) e o termo falha (consequência) a um comportamento incorreto de um programa externado de alguma forma ao usuário ou desenvolvedor.

analisadas contêm pelo menos um padrão de correção identificável, e por isso há muitas alterações que corrigem defeitos que não podem ser representadas por um de seus padrões propostos. Já Kim et al. (2006) verificaram que entre 19,3% a 40,3% dos *bugs* nos projetos analisados e 7,0% a 15,5% dos pares de *bug* e correção foram encontrados no banco de dados (memórias) construído. Esses resultados sugerem que nem todos os *bugs* presentes nos projetos analisados puderam ser corrigidos a partir dos padrões de correções armazenados no banco de dados, ou seja, mais padrões de correção de *bugs* podem ser identificados e extraídos.

O trabalho desenvolvido por Pan et al. (2008) deixa abertura para estudos e discussões mais profundas que visem a implementação de uma abordagem alternativa que consiga descrever melhor os *bugs* de um projeto de software e dessa forma propor uma nova padronização para correção de defeitos. Sabe-se ainda, que um bom entendimento dos padrões de correção de *bugs* proporciona uma melhor compreensão dos tipos mais frequentes de *bugs* em um software, e assim é possível criar intervenções na prática atual da Engenharia de Software a fim de reduzir ou eliminar grandes classes de *bugs* (Liu, 2013).

Levando em consideração o contexto apresentado, uma abordagem alternativa é proposta neste trabalho, com o uso de operadores de mutação para a descrição desses defeitos, e isso constitui uma forma de padrão de correção, cujo intuito muito se assemelha aos padrões propostos por Pan et al. (2008). Operadores de mutação são utilizados no teste de mutação para a introdução de defeitos no programa em teste e consequente geração de mutantes. Neste trabalho o seu uso foi diferente, visando a descrição de correção. Além disso, os operadores de mutação são específicos para a linguagem na qual o programa é desenvolvido e procuram modelar defeitos comuns especificamente para essa linguagem. Dessa forma, neste trabalho, levantam-se a seguinte questão de pesquisa:

Operadores de mutação podem ser usados para descrever correções de bugs assim como a proposta de Pan et al. (2008)?

2. OBJETIVOS

O presente trabalho apresenta um estudo exploratório, considerando a possibilidade de descrição de correção de *bugs* por meio da aplicação de um ou mais operadores de mutação. A fim de alcançar este objetivo geral, foram definidos os seguintes objetivos específicos:

- Analisar defeitos e correções reais aplicando os padrões de correção propostos por Pan et al. (2008);
- Analisar defeitos e correções reais aplicando operadores de mutação para a descrição dos *bugs*;
- Avaliar a cobertura na descrição das correções de defeitos pelo uso de operadores de mutação;
- Avaliar a cobertura na descrição das correções de defeitos pelos padrões de correção propostos por Pan et al. (2008);
- Investigar se os operadores de mutação propostos na literatura para a linguagem Java são capazes de modelar defeitos reais;
- Determinar quais operadores de mutação poderiam ser mais utilizados para modelar defeitos reais.

3. ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está organizado da seguinte forma. A revisão da literatura é apresentada no Capítulo 4. O Capítulo 5 apresenta a condução do estudo exploratório detalhando a metodologia utilizada e cada um dos seus passos, os resultados das análises para cada revisão. Por fim, no Capítulo 6, são destacados e discutidos os resultados obtidos, comparando-os com os encontrados por outros autores e avaliando a perspectiva para trabalhos futuros.

4. REVISÃO DA LITERATURA

Neste capítulo serão apresentados os conceitos relacionados diretamente a este trabalho que nos serviram de base para sua condução, tais como os padrões de correção de defeitos e operadores de mutação.

4.1. CONSIDERAÇÕES INICIAIS

Inicialmente, a partir da revisão da literatura, investigou-se diferentes abordagens empregadas para a correção de defeitos de software, e em especial, buscou-se compreender os padrões de correção propostos por Pan et al. (2008) para a posterior aplicação destes na correção dos defeitos selecionados. Adicionalmente, foram explorados e analisados trabalhos que abordaram o uso de operadores de mutação em várias linguagens de programação, a partir dos quais foi possível selecionar operadores para a descrição dos *bugs* extraídos do sistema de controle de versão.

4.2. PADRÕES DE CORREÇÃO DOS DEFEITOS

Pan et al. (2008), extraiu padrões de correção de *bugs* a partir dos dados do histórico de alterações de 7 projetos de código aberto. Segundo Pan et al. (2008), uma vez que os padrões de correção apresentam uma correção para os *bugs*, eles fornecem uma visão substancial do próprio *bug* inicial. Por exemplo, uma correção de *bug* que envolve um reparo em um *if* condicional indica que o condicional estava envolvido no *bug* antes da correção. Assim, conhecendo-se os padrões de correção de *bugs* e sua frequência nos projetos, é possível avaliar a frequência relativa dos diferentes tipos de *bugs.*

Para Liu et al. (2013) uma das principais razões para o estudo dos padrões de correção de *bugs* é que estes afetam a viabilidade e aplicabilidade da geração de correção automática a partir de relatos de falhas. Se muitas correções de *bugs* compartilham um pequeno conjunto de padrões de correção, então é viável gerar automaticamente correções baseadas nos padrões de correção comuns. Debroy e Wong (2014), também compartilham dessa opinião e propuseram a correção de *bugs* por meio da aplicação de operadores de mutação. Em sua abordagem, Debroy e Wong (2014) redefiniram algumas das condições do teste de mutação e analisaram o comportamento de 19 programas, cada um com exatamente um *bug* inicial, obtidos da aplicação de 8 operadores de mutação. Como nesse caso o teste de mutação já é aplicado a um

programa que apresenta *bug*, Debroy e Wong (2014) procuraram estabelecer uma relação entre o programa desejado e o programa mutante gerado após a aplicação de um operador de mutação, e não entre o programa inicialmente defeituoso e o programa resultante da aplicação do operador. Os resultados indicaram que 20,70% dos *bugs* foram corrigidos, entretanto, é importante ressaltar que apesar do uso de operadores de mutação, esse trabalho não propõe uma forma de descrever padrões genéricos de correção, mas sim de corrigir casos específicos de *bugs*.

Para definir um conjunto de padrões de correção de *bugs*, Pan et al. (2008) analisaram manualmente parte das alterações que corrigiram os *bugs* inicialmente de 5 projetos de código aberto, ArgoUML, Columba, Eclipse, JEdit, e Scarab. A análise envolveu a inspeção dos *bug hunks* e dos correspondentes *fix hunks* nas revisões de correção de *bugs*, e a classificação das alterações que corrigiram os *bugs* em diferentes padrões (tipos de *bug*).

O *bug hunk* indica uma seção do código na versão defeito que é modificado ou excluído na revisão de correção de *bug*, e seu correspondente *fix hunk* indica o código na versão de correção que corrige o defeito. Um par *bug fix hunk* refere-se, portanto, ao código fonte do defeito e também à alteração que corrige tal defeito (PAN et al., 2008), conforme pode ser observado na Figura 1. É mostrado na figura um exemplo de código fonte retirado de um dos projetos analisados. O código marcado por um "-" é proveniente do *bug hunk* (remoção), enquanto o código com "+" (adição) está no *fix hunk*. Um código não marcado é encontrado tanto na versão *bug* como na versão de correção.

```
     - lastChunk.init(seg,expander,x,styles,
     -       fontRenderContext,
context.rules.getDefault());
     + if (!lastChunk.initialized)
     +       lastChunk.init(seg,expander,x,styles,
     +       fontRenderContext,
context.rules.getDefault());
 .....
```

FIGURA 1- Exemplo de código fonte retirado do Argo UML.

Fonte: Toward an understanding of bug fix patterns (PAN et al., 2008)

Posteriormente, foi desenvolvida uma ferramenta capaz de extrair padrões de correção de *bugs* automaticamente. Essa ferramenta foi usada para analisar o histórico de alterações de 7 projetos de código aberto, incluindo 2 (Lucene e MegaMek) que não foram utilizados para desenvolver a taxonomia inicial de padrões (PAN et al., 2008).

Esses padrões de correção de *bugs* foram agrupados em várias categorias, conforme pode ser constatado na Tabela 1, que apresenta uma lista com os padrões de correção de *bugs* observados por Pan et al. (2008) nos projetos analisados:

TABELA 1-Padrões de Correção de defeitos introduzidos por Pan et al. (2008).

Categoria	**Nome do Padrão**	**Nome Abreviado**
Assignment (AS)	Alteração da expressão de *assignment*	AS-CE
Class Field (CF)	Adição de uma *class field*	CF-ADD
	Alteração da declaração de *class field*	CF-CHG
	Remoção de uma *class field*	CF-RMV
If related (IF)	Adição de um ramo *else*	IF-ABR
	Adição de verificação de pré-condição	IF-APC
	Adição de verificação de pré-condição com *jump*	IF-APCJ
	Adição de verificação de pós-condição	IF-APTC
	Alteração da expressão de condição *if*	IF-CC
	Remoção de um ramo *else*	IF-RBR
	Remoção de um predicado *if*	IF-RMV
LOOP (LP)	Alteração da condição loop	LP-CC
	Alteração da expressão que modifica a variável *loop*	LP-CE
Method Call (MC)	*Method call* com valores de parâmetros reais diferentes	MC-DAP
	Method call diferente para uma ocorrência de classe	MC-DM
	Method call com números diferentes ou com tipos de parâmetros diferentes	MC-DNP
Method Declaration (MD)	Alteração do *method declaration*	MD-CHG
	Adição de um *method declaration*	MD-ADD
	Remoção de um *method declaration*	MD-RMV
Sequence (SQ)	Adição de operações em uma Sequência de operação de *field settings*	SQ-AFO
	Adição de operações em uma sequência de *method calls* para um objeto	SQ-AMO
	Adição ou remoção de operações de *method call* em um short constructo *body*	SQ-AROB
	Remoção de operações de uma sequência de operação de *field settings*	SQ-RFO
	Remoção de operações de uma sequência	SQ-RMO

	de operação de *method calls* para um objeto	
Switch (SW)	Adição/Remoção de um ramo *switch*	SW-ARSB
TRY (TY)	Adição/Remoção de um bloco *catch*	TY-ARCB
	Adição/Remoção de uma instrução *try*	TY-ARTC

Fonte: Toward an understanding of bug fix patterns. Pan et al. (2008)

Conforme pode-se observar na Tabela 1, "Adição de verificação de pré-condição" é um dos padrões de correção de *bug* definidos por Pan et al. (2008), o qual pertence à categoria *If related* (IF), e que é responsável por adicionar um predicado *if* para garantir que a pré-condição seja encontrada antes de um objeto ser acessado ou uma operação seja realizada, a verificação da pré-condição pode ocorrer o lançamento de exceção NullPonterException ou a execução de uma operação inválida causada pelo código defeituoso. Um exemplo em que o padrão IF-APC pode ser aplicado para descrever um *bug* pode ser observado na Figura 2.

```
− lastChunk.init(seg,expander,x,styles,
−            fontRenderContext, context.rules.getDefault());
+ if (!lastChunk.initialized)
+       lastChunk.init(seg,expander,x,styles,
+            fontRenderContext, context.rules.getDefault());
```

FIGURA 2 – Exemplo de aplicação do padrão de correção IF-APC.

Fonte: Toward an understanding of bug fix patterns (PAN et al., 2008).

Conforme descrito, Kim et al. (2006) propuseram uma abordagem para encontrar *bugs*, baseada na memorização de padrões de correção de *bugs*, o que permite o uso destes padrões para encontrar *bugs* em outros projetos ou outras partes do código fonte. Kim et al. (2006) focaram seu estudo em *bugs* específicos de cada projeto, já que diferentes projetos têm diferentes exigências, convenções e abstrações.

Inicialmente Kim et al., (2006) construiu um banco de dados com as correções de *bugs* e então utilizou uma ferramenta chamada BugMem, a qual por meio das memórias construídas, detecta *bugs* específicos de projetos e sugere correções. A construção das memórias foi realizada utilizando a identificação no histórico do projeto de software, das alterações relacionadas à correção de um *bug*.

Para cada projeto específico, Kim et al. (2006) aplicaram um algoritmo que extraiu os padrões de sintaxe, chamados de componentes, de cada trecho do código fonte relacionado a uma determinada alteração de correção de *bug*, os quais foram salvos no banco de dados das memórias. A ferramenta BugMem fornece exemplos de

correção reais para os *bugs* identificados em novas alterações, usando dados de correção provenientes das memórias de correção de cada projeto.

A Figura 3 apresenta um exemplo de *output* da ferramenta BugMem, o qual indica que o uso de `setSelectText()` pode ser um *bug* potencial, e recomenda alterá-lo para `insertTab()`.

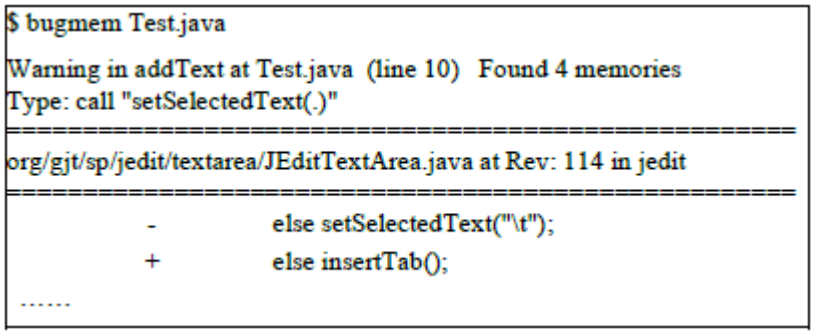

FIGURA 3 - Exemplo de output da ferramenta BugMem.

Fonte: Memories of Bug Fixes (KIM et al., 2006)

Apesar da abordagem explorada por Kim et al (2006) ser válida, os resultados obtidos a partir deste trabalho serão comparados aos resultados alcançados por Pan et al. (2008), já que a cobertura dos defeitos por meio de padrões de correção foi maior do que através do uso de um banco de dados com correções para *bugs*.

4.3. OPERADORES DE MUTAÇÃO

O teste de mutação consiste na geração de variações do programa original por meio de um conjunto de operadores de mutação, os quais provocam pequenas alterações sintáticas do programa original, transformando-os em programas mutantes (JIA e HARMAN, 2011). Este conjunto generaliza os erros de programação típicos e é determinado de acordo com a linguagem de programação utilizada para o desenvolvimento do programa em teste. Para a programação procedural, um operador de mutação pode ser responsável, por exemplo, por substituir um operador aritmético no programa original por outras alterações corretas sintaticamente (CHEVALLEY, 2001). Um exemplo de aplicação de operador de mutação para a linguagem Java pode ser verificado na Figura 4.

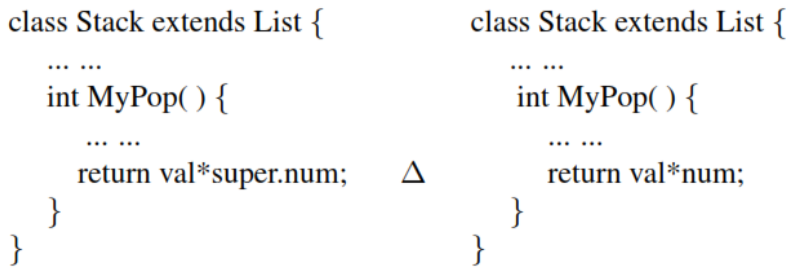

FIGURA 4- Aplicação do operador de mutação ISK (*super* keyword deletion).

Fonte: Inter-class Mutation Operators for Java (MA et al., 2002)

Segundo Ma et al. (2002), a eficácia do teste de mutação está fortemente relacionada à capacidade dos operadores de mutação em representar *bugs* de programação potenciais. Para Jia e Harman (2011), os operadores de mutação tradicionais, como por exemplo, operadores de exclusão, duplicação ou inserção, de substituição de algumas operações aritméticas por outras, não são suficientes para testar linguagens de programação orientadas a objetos como Java. Isto ocorre principalmente porque as falhas representadas pelos operadores de mutação tradicionais são diferentes daquelas presentes no ambiente orientado a objeto, devido a diferente estrutura de programação dos programas orientados a objetos. Além disso, existem novas falhas, introduzidas por recursos específicos da programação orientada a objetos, como herança e polimorfismo.

Kim et al. (1999) foram os primeiros a desenvolver operadores de mutação para a linguagem de programação Java. Eles propuseram um conjunto de 20 operadores de mutação para Java usando HAZOP (Estudos de Perigos e Operabilidade). HAZOP é uma técnica de segurança que investiga e registra o resultado de desvios do sistema. Kim et al. (1999) aplicaram HAZOP a fim de identificar as falhas plausíveis da linguagem de programação Java. Baseado nessas falhas plausíveis, 20 operadores de mutação foram projetados, os quais foram classificados em 6 grupos: tipos/variáveis, nomes, classes/declarações de interface, blocos, expressões e outros.

Baseado em seu trabalho anterior (KIM 1999) sobre os operadores de mutação Java, Kim et al. (2000) introduziram uma técnica chamada mutação de classe, que aplica mutação a programas orientados a objetos visando as falhas relacionadas aos recursos específicos desses programas. A técnica pode ser utilizada em si mesmo como uma forma de teste de mutação seletiva direcionada aos programas orientados a objetos ou pode ser integrada com os sistemas de mutação convencional.

Chevalley (2001) propôs uma extensão da Mutação de Classe com 5 novos operadores para a linguagem Java, os quais são baseados em recursos diferentes daqueles visados pelos 13 operadores de Mutação de Classe propostos por Kim et al. (2000). Entretanto, de acordo com Ma et al. (2002), as falhas modeladas por esses operadores não são gerais, pois elas podem ser específicas da aplicação ou específicas do programador. Assim, para que se possa executar teste de mutação com esses operadores, eles devem ser selecionados baseados na característica do programa a ser testado.

Ma et al. (2002) também definiram um conjunto de 25 operadores de mutação para testar as falhas identificadas no uso de recursos da programação orientada a objetos com Java, já que a maior parte dos operadores de mutação foram desenvolvidos para programas procedurais.

Esses operadores de mutação definidos por Ma et al. (2000) foram baseados nos operadores previamente definidos por (KIM et al., 1999), (KIM et al., 2000) e (CHEVALLEY et al., 2001), e em uma extensa lista de falhas de programas orientados a objetos, especificamente, falhas de nível interclasse, que ocorrem devido ao uso de recursos de linguagem como controle de acesso, herança, polimorfismo e sobrecarga.

A fim de se obter um conjunto de operadores de mutação capazes de descrever os defeitos selecionados para análise, investigaram-se também os operadores de mutação propostos para linguagens como C, ADA e FORTRAN, pois estes também são aplicáveis para a linguagem Java. É o caso do trabalho desenvolvido por King e Offutt et al. (1991), os quais apresentaram um conjunto de 22 operadores de mutação utilizados pela *Mothra.* A *Mothra* é uma ferramenta de apoio ao critério análise de mutantes para o teste de programas na linguagem FORTRAN e que apresenta interface baseada em janelas, facilitando a visualização de informações.

Os operadores de mutação utilizados pela *Mothra* podem ser divididos em três grandes classes: *statement analysis* (sal), *predicate analysis* (pda) *e coincidental correctness* (cca). Os operadores de *statement analysis* verificam se há vários tipos de erros e incluem, por exemplo, a substituição do TRAP, que garante que cada *statement* é alcançado, e a substituição do CONTINUE, a qual garante que cada *statement* é necessário para a execução correta do programa. Já os operadores de *predicate analysis* representam os erros que os programadores cometem dentro das expressões usando um

operador de comparação incorreto ou o operador aritmético errado. E os operadores de *coincidental correctness* representam os casos nos quais os programadores utilizam o nome da variável errado ou uma referência *array* (KING e OFFUTT, 1991).

Offutt et al. (1996) definiram um conjunto de 65 operadores de mutação para linguagem de programação ADA. Esses operadores de mutação foram categorizados utilizando critérios sintáticos de forma adequada tanto para um implementador de um sistema baseado em mutação, quanto para um testador que pretenda compreender como a análise de mutação pode ser usada para testar programas Ada. Além disso, Offutt *et al*. (1996) foram capazes de definir operadores de mutação para todos os elementos sintáticos de Ada, incluindo manipuladores de exceção, *generics packages* e *tasking*.

Agrawal et al. (1989) propuseram um conjunto de operadores de mutação para a linguagem de programação C, os quais foram categorizados por meio do critério sintático em: *statement mutations*, *operator mutations*, *variable mutations* e *Constant mutations*. Os operadores de mutação pertencentes a essas categorias foram delineados para modelar erros cometidos por programadores na seleção de identificadores e constantes enquanto formulam expressões, na composição de funções de expressões e na composição de funções usando declarações iterativas e condicionais.

Delamaro et al. (2001) avaliaram um critério inter procedimental baseado em Mutação chamado Mutação de Interface. Eles investigaram por meio de um conjunto de operadores de mutação se esse critério era adequado para o uso durante testes de integração. Foram definidos dois grupos de operadores de interface, num total de 33 operadores, para a linguagem de programação C. Considerando a conexão de F-G entre as funções F e G, operadores do primeiro grupo são aplicados dentro do *body* da função G. Enquanto os operadores do segundo grupo são aplicados às chamadas da função G dentro de F.

Baseados em estudos sobre o operador *Statement Deletion* (SDL) de (KING 1991), Delamaro, Offutt e Ammann et al. (2014) desenvolveram 4 novos operadores de mutação para a linguagem C e verificaram que os mesmos apresentaram um bom desempenho e podem ser considerados, dessa forma, mais evoluídos.

Na Tabela 2 encontram-se os trabalhos que foram consultados para a definição do conjunto de operadores de mutação a serem utilizados para a verificação e validação da estratégia proposta. Dos 408 operadores de mutação avaliados, 197 foram propostos para a linguagem Java e o restante, 211, para outras linguagens.

TABELA 2 - Trabalhos consultados e os respectivos números de operadores de mutação propostos.

Artigo	**Números de operadores de mutação**
The Rigorous Generation of Java Mutation Operators Using HAZOP (KIM et al., 1999)	20
Class mutation: Mutation testing for object-oriented programs (KIM et al., 2000)	13
Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach (CHEVALLEY et al., 2001)	18
Inter-class Mutation Operators for Java (MA et al.,2002)	25
MuJava: An Automated Class Mutation System (MAL et al.,2005)	24
Investigating the Effectiveness of Object-Oriented Testing Strategies Using the Mutation Method (KIM et al.,2001)	20
Evaluation of Mutation Testing for Object-Oriented Programs (MA et al.,2006)	30
A Mutation Analysis Tool for Java Programs (CHEVALLEY et al., 2003)	19
Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation (KIM et al., 1999)	3
Mutation Operators for Concurrent Java (BRADBURY et al., 2006)	25
A Fortran Language System for Mutation-Based Software Testing (KING e OFFUTT, 1991)	22
Mutation Operators for Ada (OFFUTT et al., 1996)	65
Design of Mutant Operators for the C Programming Language (AGRAWAL et al., 1989)	87
Interface Mutation: An Approach for Mutation Integration Testing (DELAMARO et al., 2001)	33
Designing Deletion Mutation Operators (DELAMARO, OFFUTT, AMMANN, 2014)	4
Total	408

Fonte: Elaborada pela autora.

5. CONDUÇÃO DO ESTUDO EXPLORATÓRIO

Neste capítulo será apresentado o método a ser aplicado, escolhendo-se o projeto JEdit.

Nas próximas seções, serão apresentadas as considerações sobre a aplicação de cada passo do método, desde a identificação dos operadores de mutação até os resultados atingidos.

5.1. PASSOS PARA A CONDUÇÃO DO ESTUDO

Na Figura 5 é apresentado um esquema com os passos adotados por este trabalho para a análise da adoção de operadores de mutação para a descrição de *bugs*. O estudo foi executado com 7 passos, conforme apresentado nas seções a seguir.

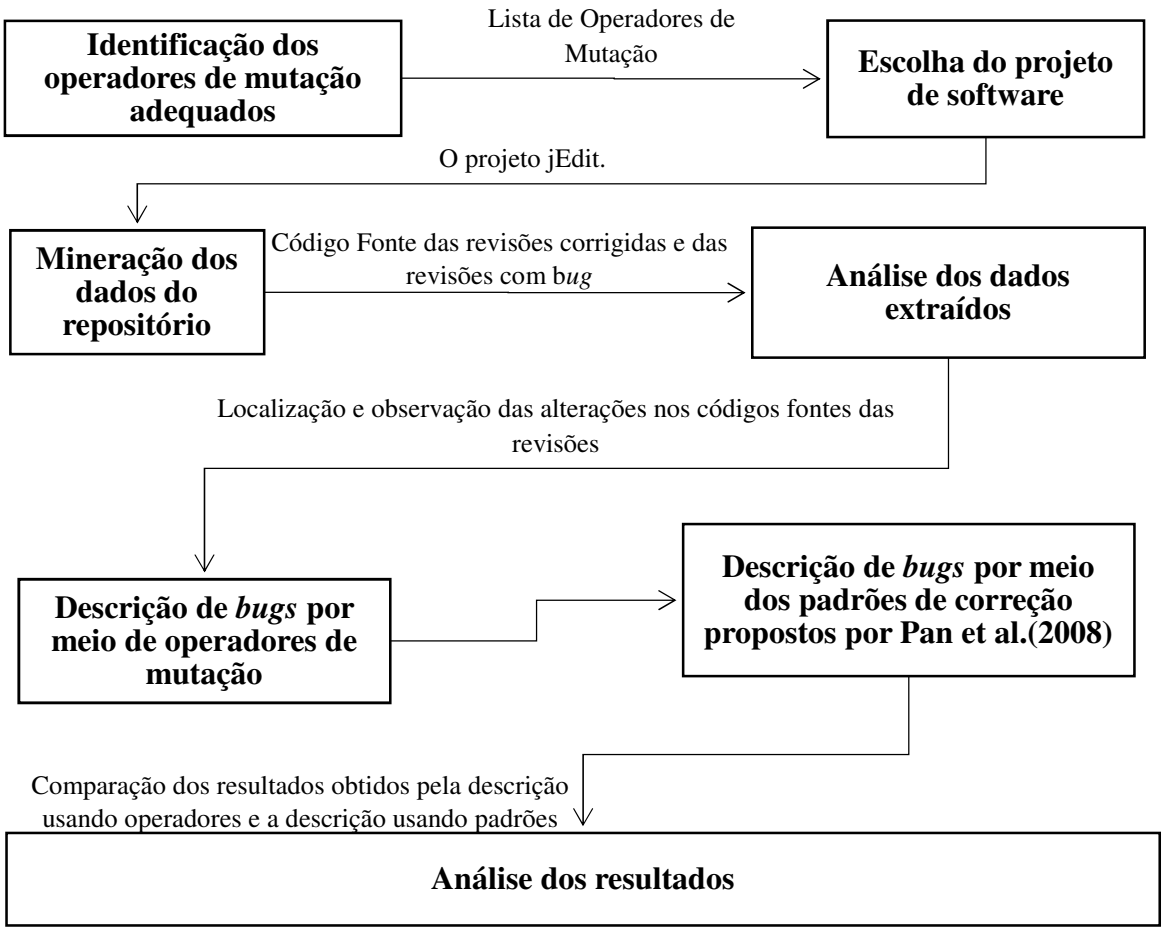

FIGURA 5- Esquema da metodologia adotada

Fonte: Elaborada pela autora.

5.2. IDENTIFICAÇÃO DOS OPERADORES DE MUTAÇÃO ADEQUADOS

A fim de identificar os operadores de mutação adequados ao trabalho em questão, realizou-se uma revisão na literatura especializada, explorando não apenas os operadores propostos para a linguagem Java, mas também para linguagens como C, ADA e FORTRAN, pois tais operadores abrangem casos não contemplados por operadores específicos para a linguagem Java.

Foi produzido um documento de trabalho com todos os operadores de mutação contidos na Tabela 2.

5.3. ESCOLHA DO PROJETO DE SOFTWARE

O projeto de software escolhido foi o jEdit, que é um editor de texto para programadores, escrito em Java e disponível sob a General Public License (GNU). Esse projeto foi selecionado a partir da lista de projetos utilizados por Pan et al. (2008), o qual já utilizou projetos com condições mínimas para a execução do processo de mineração.

5.4. MINERAÇÃO DOS DADOS DO REPOSITÓRIO

A próxima etapa compreende a mineração dos dados do sistema de controle de versão. Um sistema de controle de versão é responsável por registrar as alterações realizadas durante o desenvolvimento e manutenção de um sistema de software. Quando um desenvolvedor recebe um relato de *bug* ou pedido de modificação, ou identifica um *bug* no código fonte durante a inspeção de código, ele altera o código para corrigir o *bug* e submete as mudanças no repositório do software com uma breve nota que descreve as modificações realizadas. A versão antes da revisão de correção do *bug* é a versão *bug*, e a versão depois da revisão de correção do *bug* é a versão de correção (PAN et al., 2008).

Os elementos que compõem o arquivo de *log* do jEdit aparecem da seguinte forma: **(i)** número da revisão; **(ii)** colaborador responsável por aquela revisão; **(iii)** data; **(iv)** horário; **(v)** dia da semana em que a revisão foi realizada; **(vi)** relação de arquivos do projeto que sofreram alguma alteração; **(vii)** descrição do que foi executado. Essa estrutura pode ser visualizada na figura a seguir:

```
r3849 | spestov | 2001-10-22 09:50:57 -0200 (Seg, 22 Out 2001) | 2 linhas
Caminhos mudados:
   M /jEdit/trunk/doc/CHANGES.txt
   M /jEdit/trunk/doc/TODO.txt
   M /jEdit/trunk/org/gjt/sp/jedit/Buffer.java
   M /jEdit/trunk/org/gjt/sp/jedit/buffer/ContentManager.java
   M /jEdit/trunk/org/gjt/sp/jedit/buffer/OffsetManager.java
   M /jEdit/trunk/org/gjt/sp/jedit/gui/StatusBar.java
   M /jEdit/trunk/org/gjt/sp/jedit/io/BufferIORequest.java
   M /jEdit/trunk/org/gjt/sp/jedit/jEdit.java
   M /jEdit/trunk/org/gjt/sp/jedit/jedit_gui.props
   M /jEdit/trunk/org/gjt/sp/jedit/textarea/FoldVisibilityManager.java
   M /jEdit/trunk/org/gjt/sp/jedit/textarea/MarkerHighlight.java
   M /jEdit/trunk/org/gjt/sp/util/IntegerArray.java
   M /jEdit/trunk/org/gjt/sp/util/ReadWriteLock.java

Fixed a number of content manager bugs
```

FIGURA 6- Estrutura do arquivo de log do projeto jEdit.

Fonte: Sistema de Controle de Versão do jEdit.

A fim de minerar os dados do repositório do jEdit foi necessário minerar as versões defeituosas e suas respectivas versões corrigidas. A versão corrigida será denominada Q, e a versão defeituosa será chamada de P. Deve-se então investigar a diferença entre P e Q em termos de operadores de mutação, sendo que P e Q podem ter diversas classes alteradas.

Considerando a revisão r4009 (defeituosa) e r4010 (a corrigida), tem-se que:

- P é obtido por:

```
svn co svn://svn.code.sf.net/p/jedit/svn/jEdit/trunk@4009
```

- Q é obtido por:

```
svn co svn://svn.code.sf.net/p/jedit/svn/jEdit/trunk@4010
```

A seleção das versões corrigidas foi realizada por meio do arquivo de log dos *commits* inicialmente filtrando somente os relatos que continham as palavras *bug*, *fixed* ou *patch*, o que resultou nas versões de correção, sendo que a versão exatamente anterior correspondia a versão defeituosa.

5.5. ANÁLISE MANUAL DOS DADOS EXTRAÍDOS

A partir dos relatos encontrados, foram selecionados somente aqueles que envolviam códigos Java e eliminados aqueles cujas alterações ultrapassassem o total de 7 linhas mesmo parâmetro utilizado por PAN et al. (2008) modificadas dentro de um mesmo método. A seguir, a partir das revisões selecionadas, foram filtradas somente as

revisões com datas de ocorrência entre 2001 e 2006. Por fim, essas revisões filtradas foram numeradas e com o auxilio da ferramenta web random[2]realizou-se o sorteio aleatório de 20 delas. Contudo para que fosse atingido o número de 20 revisões analisáveis tivemos que sortear 25 revisões.

As revisões r5209, r6459 e r7883, por exemplo, de acordo com o critério de seleção estabelecido, foram descartadas, pois durante a análise dos códigos fonte verificou-se que ambas apresentavam alterações com mais de 7 linhas. Constatou-se ainda, que na revisão r4001 não houve modificações entre suas versões defeituosa e corrigida, ou seja, trata-se de um caso no qual havia um relato de *bug* no arquivo log, mas na prática não havia um *bug* algum na revisão, não sendo então necessária a correção. Caso semelhante ocorreu com a revisão r6681, pois a mesma também se referia a uma correção de *bug*, entretanto se tratava apenas de uma melhoria no código fonte para compatibilidade com o módulo do java 1.5 api, então, descartou-se.

5.6. DESCRIÇÃO DE *BUGS* POR MEIO DE OPERADORES DE MUTAÇÃO

Por meio da seleção dos operadores de mutação descritos por diversos autores para diferentes linguagens de programação, realizou-se uma análise cuidadosa das alterações que resultaram em correções. Para cada *bug* presente em cada revisão, houve a tentativa de encontrar uma sequência de operadores de mutação capazes de descrever essas alterações, ou seja, de corrigir o código fonte defeituoso.

Ao se investigar a aplicação de operadores para a correção dos *bugs*, levou-se em consideração aspectos como a influencia da ordem de aplicação dos operadores de mutação no resultado; a existência de algum padrão que não permitia a correção de um defeito, no caso desta não ter sido descrita pelos operadores de mutação e se haviam diferentes sequências de operadores de mutação que eram capazes de descrever a mesma alteração.

A partir dos dados obtidos realizou-se a análise das versões *bug* e corrigida de cada revisão a fim de detectar o que foi alterado. A seguir, listou-se cada uma das alterações realizadas para cada revisão e com base nisso realizou-se uma consulta na lista de operadores de mutação previamente selecionada com o intuito de identificar um

[2]www.random.org

operador de mutação ou uma combinação deles capazes de efetuar as mesmas alterações.

Em outras palavras, aplicaram-se os operadores de mutação na versão defeituosa das revisões e a seguir observou-se o código fonte com o objetivo de verificar se o estado desejado foi alcançado, ou seja, o mesmo estado alcançado pela aplicação de padrões de correção na versão corrigida. Quando a correção do *bug* foi contemplada integralmente pela aplicação dos operadores de mutação, então se considerou que nesse caso obteve-se uma correção alternativa àquela realizada por meio de padrões de correção.

5.7. ANÁLISE DOS RESULTADOS

Verificou-se que no arquivo de log do jEdit havia 19.694 relatos de revisões, dos quais 913 continham as palavras *bug*, *fixed* ou *patch* conforme a análise realizada. Dessas 913 revisões, selecionou-se 423 que envolviam códigos Java. Eliminou-se então as revisões que apresentavam mais 7 de linhas e obteve-se 413 revisões. Posteriormente, foram selecionadas as revisões com datas entre 2001 e 2006, pois essa foi a mesma faixa de datas levada em consideração por Pan et al. (2008) em seu estudo, assim foi possível obter um total de 313 relatos, dos quais foram sorteados 25 para a aplicação dos operadores. Entretanto, as revisões r5209, r6459 e r7883, de acordo com o critério de seleção estabelecido, foram descartadas, pois durante a análise dos códigos fonte verificou-se que ambas apresentavam alterações com mais de 7 linhas.

Por fim, obteve-se um total de 20 revisões, sendo elas: r4010, r4235, r4445, r4535, r5222, r5247, r5255, r5280, r5606, r6682, r6923, r7490, r8090, r8107, r8247, r8282, r7490, r5452, r5222, r4004.

Para obter as revisões utilizou-se o comando *svn checkout* do svn. Inicialmente foi criado um diretório chamado jEdit, dentro do qual foram criados dois novos diretórios sendo um chamado *bug* e o outro chamado *bugfixed* para cada versão selecionada. E, então, foram executados dois comandos svnco (svn checkout) para os diretórios *bug* e *bugfixed* de cada versão. Considerando a revisão chamada R8108, assim tem-se como exemplo:

- no diretório jedit\R4009*bugfixed* foi executado o comando:

```
svn co svn://svn.code.sf.net/p/jedit/svn/jEdit/trunk@4009
```

- no diretório jedit\R8108*bug* foi executado o comando:]

```
svn co svn://svn.code.sf.net/p/jedit/svn/jEdit/trunk@8107
```

Conforme foi apresentado, a partir das alterações realizadas no código fonte em cada revisão para a correção dos *bugs*, foi possível verificar se um operador ou um conjunto de operadores de mutação também poderiam descrever, ou seja, corrigir esses defeitos, da mesma forma que os padrões de correção de defeitos apresentados por Pan et al. (2008).

A relação de revisões analisadas e os operadores utilizados para a descrição de cada alteração encontrada no código fonte encontram-se relacionadas em uma tabela anexa a reste trabalho no apêndice A.

6. DISCUSSÃO

Neste capítulo será apresentada a discussão referente aos resultados obtidos com aplicação dos conceitos estudados de operadores de mutação e padrões de correção de defeitos.

6.1. OPERADORES DE MUTAÇÃO

De acordo com os resultados mostrados na Tabela 3, verificou-se que por meio da aplicação dos operadores de mutação, nas 20 revisões selecionadas, foi possível corrigir somente uma integralmente, sendo que 12 revisões foram corrigidas parcialmente e 7 não puderam ser corrigidas.

A Figura 7 ilustra esse comportamento, apresentando a porcentagem de cobertura por meio de operadores de mutação para cada uma das revisões selecionadas.

FIGURA 7 - Correção por Operadores de Mutação.

Fonte: elaborado pela autora.

Por meio da análise da Figura 7, é possível inferir ainda que das 12 revisões corrigidas parcialmente, apenas 2 apresentaram cobertura de 75% por operadores de mutação, ou seja, mesmo dentre as revisões corrigidas parcialmente, a porcentagem de cobertura por operadores de mutação ainda foi baixa. Além disso, observou-se que 65% (13) das revisões puderam ser descritas por pelo menos um operador de mutação do conjunto previamente selecionado.

6.2. PADRÕES DE CORREÇÃO

Baseado na categorização dos padrões de correção proposta por Pan et al. (2008), foi possível verificar a correção das revisões sorteadas por meio da aplicação desses padrões de correção previamente descritos. A Figura 8 a seguir apresenta a porcentagem de cobertura das revisões analisadas por padrões de correção de *bugs*.

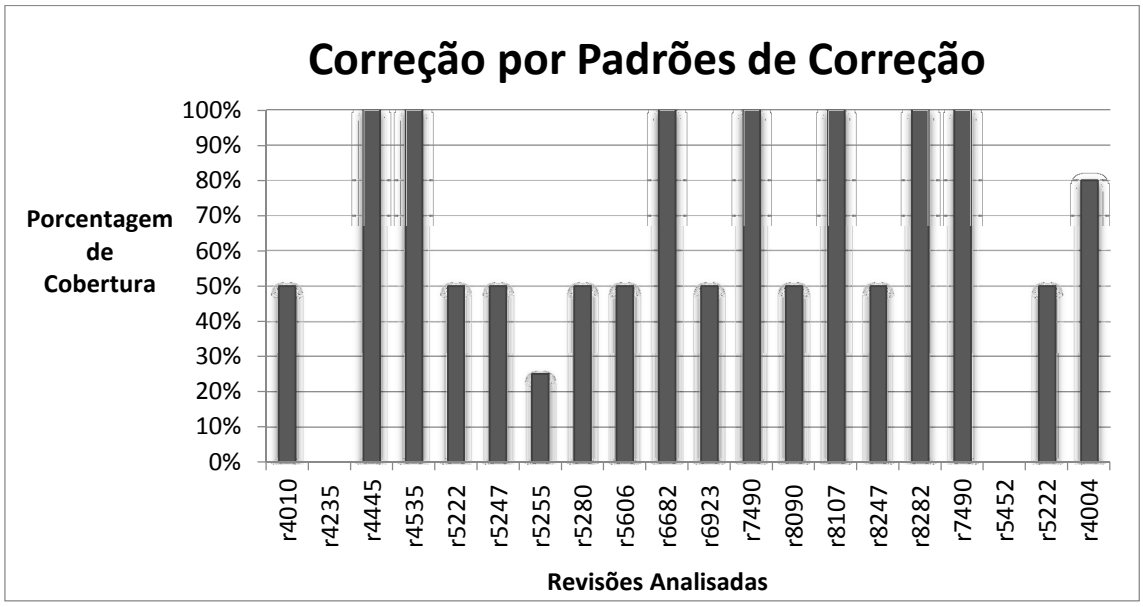

FIGURA 8 - Correção por Padrões de Correção.

Fonte: elaborado pela autora.

De acordo com a Figura 8, verifica-se que das 20 revisões analisadas, 7 puderam ser corrigidas completamente por meio dos padrões de correção de defeitos definidos por Pan et al. (2008), 11 revisões foram corrigidas parcialmente e 2 não puderam ser corrigidas. Foi possível observar ainda que 90% (18) das revisões analisadas continham pelo menos um padrão de correção proposto por Pan et al., (2008).

Este resultado foi superior ao encontrado por Pan et al.(2008), que por meio da mineração dos dados do histórico de alterações do projeto jEdit utilizando uma ferramenta extratora, verificou que aproximadamente 63,6% das alterações analisadas continham pelo menos um padrão de correção identificável.

6.3. OPERADORES DE MUTAÇÃO *versus* PADRÕES DE CORREÇÃO

A partir dos resultados anteriores sobre a porcentagem de cobertura dos operadores de mutação e dos padrões de correção definidos por Pan et al. (2008), foi possível estabelecer uma comparação entre as duas formas de descrever os defeitos das revisões selecionadas. A Figura 9 ilustra essa comparação e apresenta para cada revisão a porcentagem de cobertura tanto para a correção por meio de operadores de mutação como por padrões de correção.

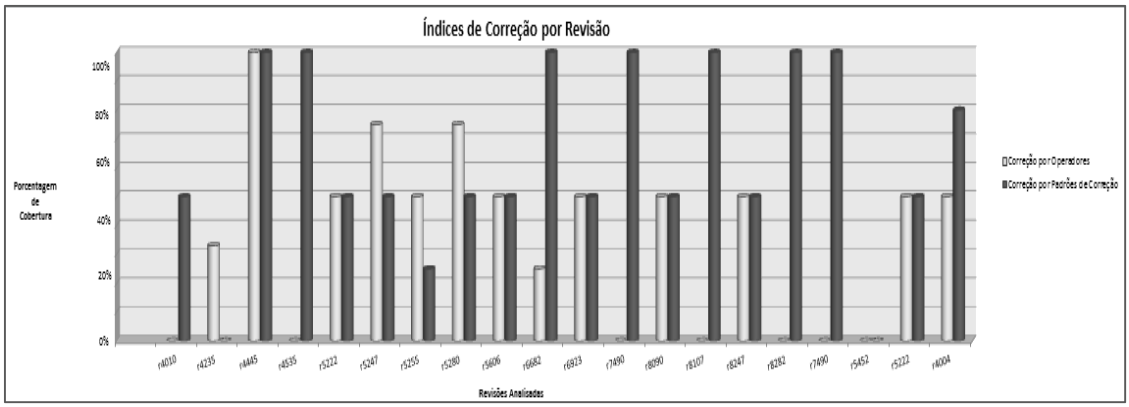

FIGURA 9 - Comparação das porcentagens de coberturas por operadores de mutação e padrões de correção para cada revisão.
Fonte: elaborado pela autora.

A partir da análise da Figura 9 se verifica que para as mesmas 20 revisões, os padrões de correção propostos por Pan et al. (2008) foram mais eficientes na correção dos defeitos do que os operadores de mutação previamente selecionados. É possível observar ainda que em 80% das revisões a porcentagem de cobertura por padrões de correção foi maior ou igual a porcentagem de cobertura dos operadores de mutação.

A Figura 10 apresenta uma média da porcentagem de cobertura das revisões analisadas por operadores de mutação e por padrões de correção.

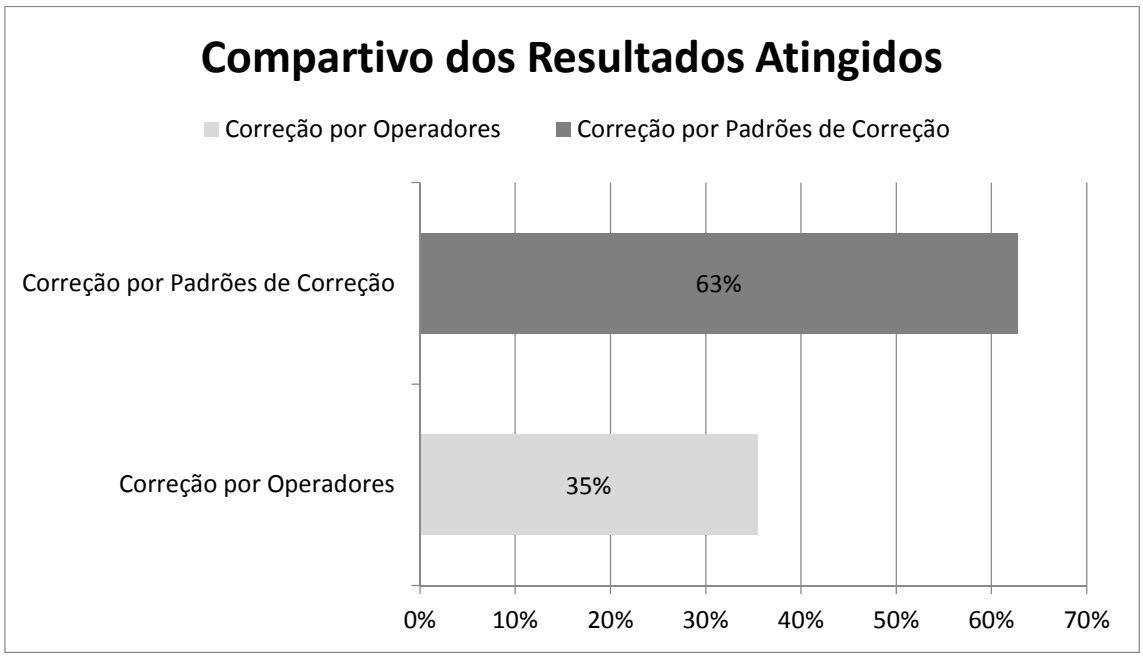

FIGURA 10 - Porcentagem de cobertura média das revisões por Operadores de Mutação versus Padrões de Correção.

Fonte: elaborado pela autora.

Ao observar a Figura 10, pode-se ver que é evidente que os padrões de correção apresentam uma considerável porcentagem de cobertura das revisões quando comparados aos operadores de mutação, conforme já discutido anteriormente.

6.4. ANÁLISE QUALITATIVA

A aplicação de um ou mais operadores de mutação para a correção de defeitos mostrou-se como uma técnica útil, no sentido que esses operadores foram capazes de descrever algumas das alterações das revisões selecionadas. No entanto, verificou-se que os padrões de correção apresentaram uma porcentagem de cobertura média de 64% em relação a 36% dos operadores de mutação.

Entretanto, é importante destacar que Pan et al. (2008) selecionou e analisou suas revisões automaticamente por meio de uma ferramenta, isto é, nem todas as revisões analisadas por ele podem ser realmente defeituosas e nesse sentido a abordagem explorada nesse trabalho pode ser considerada mais consistente, já que a análise foi feita de maneira manual e dessa forma é possível garantir que trata-se de defeitos reais. Conforme foi apresentado anteriormente, houve revisões que foram descartadas por se tratarem de melhorias ou por não haver diferença entre o código fonte defeituoso e o corrigido, e a ferramenta extratora utilizada por Pan et al. (2008)

pode não ter sido realmente eficiente na análise desse tipo de ocorrência, e portanto, os resultados obtidos pelo autor podem ser contestáveis.

Além disso, houve 4 revisões (r4235, r5247, r5255, r5280) que apresentaram porcentagem de cobertura por meio de operadores de mutação maior do que por padrões de correção, o que demonstra que é possível descrever *bugs*, inclusive integralmente, por meio da aplicação de operadores de mutação, e assim, propor novos padrões para a correção de defeitos de software por meio do uso de operadores.

7. CONCLUSÃO

Este trabalho buscou a descrição de defeitos de software empregando o uso de operadores de mutação, onde o objeto de estudo foi o software de código aberto JEdit.

Em relação aos objetivos definidos para este trabalho, obtivemos os seguintes resultados.

Por meio de uma análise do arquivo de *log* do jEdit, o qual foi obtido através do sistema de controle de versão do projeto, tornou-se possível selecionar algumas revisões, as quais puderam ser analisadas por meio da aplicação de operadores de mutação e dos padrões de correção descritos por Pan et al. (2008).

Para cada revisão obteve-se a porcentagem de cobertura tanto por operadores de mutação como por padrões de correção, resultado esse que levou em consideração o número de linhas do código fonte contempladas com as correções, sendo que estes últimos apresentaram uma porcentagem de cobertura média de 64% em relação a 36% dos operadores de mutação, o que indica que o conjunto de operadores de mutação selecionados a partir de uma revisão da literatura especializada não foi suficiente para a descrição dos defeitos presentes nas revisões sorteadas. Esses resultados sugerem que ainda existe potencial para trabalhos futuros, a fim de desenvolver novos operadores de mutação para a descrição dos mais diversos *bugs* nas várias linguagens de programação, para que se possa obter então uma melhor cobertura dos defeitos.

Por fim, foi realmente possível descrever os *bugs* de um projeto de software pelo uso de operadores de mutação, ou seja, a abordagem investigada no trabalho mostrou-se útil, sendo importante ressaltar a validade da mesma já que os resultados foram obtidos para defeitos reais, os quais foram avaliados através de uma análise manual minuciosa.

As limitações do estudo com o projeto JEdit se consolidaram com fato de que os operadores de mutação não contemplam correções que possuam inserção de código. A caracterização deste problema tornou-se evidente em nossa análise, talvez este problema seja contornado com o surgimento de novos operadores.

Os trabalhos futuros incluem analisar os resultados atingidos por meio de ferramentas estatísticas a fim de gerar melhor confiabilidade ao estudo. Outro ponto a ser investigado é a forma de representação dos resultados obtidos neste trabalho.

REFERÊNCIAS

AGRAWAL, Hiralal et al. **Design of Mutant Operators for the C Programming Language.** Software Engineering Research Center, Purdue University, West Lafayette, 1989.

BRADBURY, Jeremy S.; CORDY, James R; DINGEL, Juergen. **Mutation Operators for Concurrent Java.** Proceedings of the Second Workshop on Mutation Analysis, Washington, 2006.

DEBROY, Vidroha; WONG, Eric. **Using Mutation to Automatically Suggest Fixes for Faulty Programs.** Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST), Paris, França, p. 65-74, 2010.

DEBROY, Vidroha; WONG, W. Eric. **Combining Mutation and Fault Localization for Automated Program Debubing.** The Journal of Systems and Software, 2014.

CHEVALLEY, Philippe. **Applying mutation analysis for object oriented programs using a reflective approach.** Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001), Macau SAR, China, December 2001.

CHEVALLEY, P.; THÉVENOD-FOSSE, P. **A mutation analysis tool for Java programs.** International Journal on Software Tools for Technology Transfer. v. 5, n. 1, p. 90-103, 2003.

DELAMARO, Márcio E.; MALDONADO, José A.; MATHUR, Aditya P. **Interface Mutation: An Approach for Integration Testing.** IEEE Transactions on Software Engineering, v. 27, n. 3, 2001.

DELAMARO, Márcio E.; OFFUTT, Jefferson; AMMANN, Paul. **Designing Deletion Mutation Operators.** Proceedings of the 2014 IEEE International Conference on Software Testing, Verification and Validation, p. 11-20, Washington, USA, IEEE Computer Society, 2014.

JIA, Yue; HARMAN, Mark. **An Analysis and Survey of the Development of Mutation Testing.** Software Engineering, IEEE Transactions on. v. 37, n. 5, p. 649-678, 2011.

KIM, Sunwoo; CLARK, John A.; MCDERMID, John A. **Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation**. Proceedings of the 28th JAIIO: Symposium on Software Technology (SoST'99), Buenos Aires, 1999.

KIM, Sunwoo; CLARK, John A.; MCDERMID, John A.**The Rigorous Generation of Java Mutation Operators Using HAZOP**. Proceedings of the 12th International Conference Software and Systems Engineering and their Applications (ICSSEA 99), Paris, France, 29 November-1 December 1999.

KIM, Sunwoo; CLARK, John A.; MCDERMID, John A. **Class mutation: Mutation testing for object-oriented programs.** Proceedings of the Conference on Object-Oriented Software Systems, 2000.

KIM, Sunwoo; CLARK, John A.; MCDERMID, John A. **Investigating the effectiveness of object-oriented testing strategies using the mutation method.** Software Testing Verification and Reliability, v. 11, p. 207-225, 2001.

KIM, Sunghun; PAN, Kai; WHITEHEAD JR, E. James**. Memories of Bug Fixes.** Proceedings of the 14th International Symposium on Foundations of Software Engineering.New York, p. 35-45, 2006.

KIM, Sunghun; WHITEHEAD Jr, E. James; ZHANG, Yi. **Classifying Software Changes: Clean or Buggy?** The IEEE Transactions on Software Engineering, v. 34, n. 2, p. 181-196, 2006.

KING, K. N.; OFFUTT, Jefferson A.**A FORTRAN Language System for Mutation-Based Software Testing.** Journal Software – Practice & Experience, v. 21, n. 7, p. 685-718, 1991.

LI, Zhenmin; et al. **Have Things Changed Now?** Proceedings of the 1 st workshop on Architectural and system support for improving software dependability**.** New York, p. 25-33, 2006.

LIU, Chen; YANG, Jingiu; TAN, Lin; HAFIZ, Munawar. **R2Fix: Automatically Generating Bug Fixes from Bug Reports.** IEEE Fifth International Conference on Software Testing, Verification and Validation, p. 282-291, 2013.

MA, Yu-Seung; OFFUTT, Jeff; KWON, Yong Rae.**Inter-class Mutation Operators for Java.**Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02). Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, p. 352.

MA, Yu-Seung; HARROLD, Mary Jean; KWON, Yong-Rae. **Evaluation of Mutation Testing for Object Oriented Programs.** Proceedings of the 28th International Conference on Software Engineering, New York, p. 869-872, 2006.

MAL, Yu-Seung; OFFUTT, Jeff; KWONL, Yong Rae. **MuJava: an automated class mutation system.** Software Testing, Verification and Reliability, v. 15, p. 97-133, 2005.

MALDONADO, José Carlos; et al. **Introdução ao Teste de Software,** 2007.

NETTO, Fernando de Castro; BARROS, Márcio de Oliveira; BAIÃO, Fernanda Araujo. **Mineração da Base de Dados de Defeitos de Software.** Relatórios Técnicos do Departamento de Informática Aplicada da UNIRIO. Universidade Federal do Estado do Rio de Janeiro. Rio de Janeiro, 2009.

OFFUTT, Jefferson; et al. **Mutation Operators for Ada.** Technical Report ISSE-TR-96-09, 1996.

PAN, Kai; KIM, Sunghun; WHITEHEAD JR; E. James. **Toward an understanding of bug fix patterns.** Empirical Software Engineering, v. 14, p. 286-315, 2008.

JUST, René; KAPFTHAMMER, Gregory M; SCHWEIGGERT, Franz. **Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?** Proceedings of the 5rd International Conference on Software Testing, Verification and Validation (ICST), Montreal, Canadá, 2012.

SUDHAKRISHNAN Sangeetha; MADHAVAN, Janaki T.; WHITEHEAD JR, E. James; RENAU, Jose. **Understanding Bug Fix Patterns in Verilog.** Proceedings of the 2008 International Working conference on Mining Software Repositories, New York, 2008.

IORA Juliana, RÉ Reginaldo; **Lista de operadores de mutação encontrados na literatura**. Documento de Trabalho, 2015 to pags 15.

APÊNDICE A – Análise das revisões defeituosas e corrigidas analisadas.

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP[3]	Cobertura PA[4]
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Complete Word	r4009	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r4010	277 SwingUtilities.invokeLater (new Runnable() 278 { 279 public void run() 280 { 281 textArea.requestFocus(); 282 } 283 });	PPD - Parameter variable declaration with child class type; PMD - Member variable declaration with parent class type; PNC - New method call with child class type.	MC-DM – Change of Method Call to a Class Instance	0%	50%

[3] OP= Operadores de Mutação.

[4] PA= Padrões de Correção de Bugs, propostos por PAN.

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
BufferPrintable	r4234	121 System.err.println(((Graphics2D)_gfx).getFontRenderContext()); 122 System.err.println(((Graphics2D)_gfx).getTransform());	r4235	113 Graphics2D gfx = (Graphics2D)_gfx; 114 gfx.setFont(font); 121 System.err.println(gfx.getFontRenderContext()); 122 System.err.println(gfx.getTransform()); 123 System.err.println(gfx.getFontMetrics().stringWidth(124 "hello world thisis a teststring!!!")); 125 System.err.println(gfx.getFont().getStringBounds(126 "hello world thisis a teststring!!!", 127 gfx.getFontRenderContext()).getWidth());	PPD - Parameter variable declaration with child class type; PMD - Member variable declaration with parent class type;	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE PADRÕES DE CORREÇÃO	33%	0%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Buffer	r4444	`2479 else` `2480lineBrackets++;`	r4445	`2479 elseif(lineBrackets>=0)` `2480lineBrackets++;`	CMC – Multiple Condition Coverage; ELR – Logical operator replacement	LP-CC – Change of Loop Predicate	100%	100%

JEditTextArea TextAreaPainter	r4534	`280 recalculateLastPhysicalLine();`
		
		`367 recalculateLastPhysicalLine();`
		
		`991`
		`992`
			r4535	`279 if(buffer.isLoaded())`
				`280 recalculateLastPhysicalLine();`
				
				`366 if(buffer.isLoaded())`
				`367 recalculateLastPhysicalLine();`
				
				`991 if(!buffer.isLoaded())`
				`992  return;`
				
				`169 if(textArea.getBuffer().isLoaded())`
					#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	IF-APC - Addition of Precondition Check;
						
						IF-APCJ - Addition of Precondition Check with Jump;
						
						IF-APTC - Addition of Post-condition Check
							0%	100%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				

JEditText Area	r5221	318
		319 selectNone();
		
		3824 if(nextFold>= -1)
		3825 {
		3826 getToolkit().beep();
		3827 return;
		3828 }
			r5222	318
				if(buffer.isLoaded())
				319 selectNone();
				
				3824 if(nextFold == -1)
				3825 {
				3826
				getToolkit().beep();
				3827 return;
				3828 }
					AOR - Arithmeticoperator replacement	IF-APC - Addition of Precondition Check	50%	50%

TextAreaTransfer Handler	r5246	207 { 208 if (selections[i].end<insertPos) 209 insertPos- =(selections[i].end- selections[i].start); 210 } 246 textArea.setSelectedText(nul l,false);	r5247	49 private static intinsertOffset; 207 { 208 if (selections[i].end<(insert Pos+insertOffset)) 209 insertOffset- =(selections[i].end- selections[i].start); 210 } 248 insertPos += insertOffset;	PMD - Member variable declaration with parent class type; CDE – Decision Coverage	IF-CC - Change of If Condition Expression; IF-SUB-AV - Addition of Variable to Condition	75%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Text Utilities	r5254	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r5255	646 if(i == str.length() – 1) 647 return i + 1;	IndVarIncDec - Inserts/removes increment and decrement operations at non interface uses; PMD - Member variable declaration	IF-APC - Addition of Precondition Check; IF-APCJ - Addition of Precondition Check with Jump;	50%	25%
JEditText Area	r5279	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r5280	767 if (screenLine< 0) 768 { 769 point = new Point(0,0); 770 screenLine = 0; 771 }	OVC - Variable replaced by a constant; SAR – Scalar variable for array reference replacement	IF-APC - Addition of Precondition Check; IF-APCJ - Addition of Precondition Check with Jump;	75%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				

VFSDirectoryEntryTableModel	r5605	44 extAttrs = new ArrayList();
		
		48 public void setRoot(VFS vfs, ArrayList list)
		
		59 files[i] = new Entry((VFSFile)list.get(i),0);
		
		79 collapse(vfs,startIndex);
		
		126 else
		127 lastIndex++;
		
		245 {
		246 Iterator iter = extAttrs.iterator();
		247 while(iter.hasNext())
		248
		249 {
		250 ExtendedAttributeattr = (ExtendedAttribute)
		251 iter.next();
		252
		if(attrs[i].equals(attr.name))
			r5606	44 extAttrs = new ArrayList<ExtendedAttribute>();
				
				48 public void setRoot(VFS vfs, List<VFSFile> list)
				
				59 files[i] = new Entry(list.get(i),0);
				
				78 if (startIndex != -1)
				79
				collapse(vfs,startIndex);
				
				126
				127 lastIndex++;
				
				245 {
				246
				247
				248 for (ExtendedAttributeattr : extAttrs)
				249 {
				250
				251
				252 if (attrs[i].equals(attr.name))
					VVDL- Variable Deletion;
					
					OVA - Variable replaced by an array reference
						LP-CC - Change of Loop Predicate	50%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Shortcuts Option Pane	r6681	87 Enumeration e = models.elements(); 88 while(e.hasMoreElements()) 89 ((ShortcutsModel)e.nextElement()).save(); 363 shortcut1 = k1[1].shortcut; 364 shortcut2 = k2[1].shortcut;	r6682	90 for (ShortcutsModel model : models) 91 model.save(); 352 if(col == 1)	IHI - Hiding Variable Insertion; SSDL - Statement Deletion	LP-CC – Change of Loop Predicate; IF-APC – Addition of Precondition Check	25%	100%
Install Panel	r6922	260 return Boolean.valueOf(entry.install); 512 table.setRowSelectionInterval(0,0);	r6923	260 return entry.install; 511 if (table.getRowCount() != 0) 512 table.setRowSelectionInterval(0,0);	RetStaRep – Replacesreturnstatement	IF-CC - Change of If Condition Expression; IF-SUB-AV - Addition of Variable to Condition	50%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
SearchAnd Replace	r7489	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r7490	1173 if (offset >= end) 1174 break loop;	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	IF-SUB-AO - Addition of Operator to Condition	0%	100%
CBZip2 InputStream	r8089	427 intzzi; 428 char thech = 0; 431 thech = (char)m_input.read(); 439 if(thech == -1	r8090	427 intzzi; 432 zzi = m_input.read(); 439 if(zzi == -1)	OVV - Variable replaced by a variable	IF-APC - Addition of Precondition Check	50%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Token Marker	r8106	344 !checkRule.upHashChar.equals(new String(line.array,pos,checkRule.upHashChar.length()))))	r8107	344 !checkRule.upHashChar.equals(new String(line.array,pos,checkRule.upHashChar.length()).toUpperCase()))	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE PADRÕES DE CORREÇÃO	0%	0%

Search Dialog	r8246	63 else
		64
		65
		66 return viewHash.get(view);
			r8247	63 else
				64{
				65
				66
				SearchDialogsearchDialog = viewHash.get(view);
				67 if (searchDialog == null)
				68 {
				69 searchDialog = new SearchDialog(view);
				70 viewHash.put(view, searchDialog);
				71 }
				72 returnsearchDialog;
				73 }
					IHI - Hiding Variable Insertion; HFA - Hiding Field variable addition; IHF - Hiding Variable Insertion; HFA - Hiding Field variable addition IHI - Hiding Variable Insertion; HFA - Hiding Field variable addition	IF-APC - Addition of Precondition Check	50%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
TextArea	r8281	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r8282	4942 else if (horizontal.getValue() != -getHorizontalOffset()) 4943 { 4944 horizontal.setValue(-getHorizontalOffset()); 4945 } 5167 protected intcaretLine;	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	IF-APC - Addition of Precondition Check	0%	100%
SearchAnd Replace	r7489	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r7490	1173 if (offset >= end) 1174 break loop;	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	IF-APC - Addition of Precondition Check	0%	100%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				
Token Marker	r5451	31 import java.util.regex.PatternSyntaxException; 34 import org.gjt.sp.util.Log; 749 private char[] substitute(Matcher match, char[] end) 860 private booleancharArraysEqual(char[] c1, char[] c2) 861 { 862 if(c1 == null) 863 return (c2 == null); 864 else if(c2 == null) 865 return (c1 == null);	r5452	340 //if(seenWhiteSpace) 341 return i; 342 //else 343 // break; 448 // if(seenWhiteSpace) 449 return i + 1; 450 // else 451 // break;	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE OPERADORES DE MUTAÇÃO	#ALTERAÇÕES NÃO PODEM SER DESCRITAS POR MEIO DO USO DE PADRÕES DE CORREÇÃO	0%	0%
JEdit TextArea	r5221	#BLOCO DE CÓDIGO NÂO EXISTIA NA VERSÂO BUG	r5222	318 if(buffer.isLoaded())	IF-APC - Addition of Precondition Check	IF-APC - Addition of Precondition Check;	50%	50%

Nome Arquivo	Versão com Bug		Versão Corrigida		Operadores de Mutação Usados	Padrões Aplicáveis	Cobertura OP	Cobertura PA
	Número Versão	Código Fonte Alterado	Número Versão	Código Fonte Alterado				

Text Utilities	r4003	340 if(seenWhiteSpace)
		341 return i + 1;
		342 else
		343 break;
		
		448 if(seenWhiteSpace)
		449 return i;
		450 else
		451 break;
			r4004	340
				//if(seenWhiteSpace)
				341 return i;
				342 //else
				343 // break;
				
				448 //
				if(seenWhiteSpace)
				449 return i + 1;
				450 // else
				451 // break;
					IF-RMV - Removal of an If Predicate;
					
					IF-RBR - Removal of an Else Branch
						IF-RMV - Removal of an If Predicate;
						IF-RBR - Removal of an Else Branch;
							50%	80%

Fonte: Elaborada pela autora