UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Marcus Adriano Ferreira Pereira

Um Estudo Empírico para Caracterização de *Benchmarks* para Reparo Automático de Software

Uberlândia, Brasil

2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Marcus Adriano Ferreira Pereira

Um Estudo Empírico para Caracterização de *Benchmarks* para Reparo Automático de Software

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Marcelo de Almeida Maia

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2020

Marcus Adriano Ferreira Pereira

Um Estudo Empírico para Caracterização de *Benchmarks* para Reparo Automático de Software

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 23 de dezembro de 2020:

Marcelo de Almeida Maia
Orientador

Fabiano Azevedo Dorça

Victor Sobreira

Uberlândia, Brasil
2020

Agradecimentos

Agradeço a Deus pela vida, oportunidade e por permitir que eu pudesse caminhar com saúde ao longo de toda a trajetória universitária.

Agradeço aos meus pais, Adriano e Daniana aos meus irmãos Gabriele, João Pedro e Francisco que são pessoas que sempre me apoiaram e prestaram todo o apoio que precisei, principalmente nos momentos mais difíceis e por serem pessoas fundamentais em minha vida.

Agradeço aos meus avós paternos Irene e Antônio Celso e maternos Rosa e Antônio Gonçalves, que sempre contribuíram com todo apoio, carinho e muito amor.

Agradeço aos meus tios Luis Antônio e Elani, Mauro e Liuziane e aos meus primos Yvens e Maycon, que sempre me apoiaram e ajudaram durante a minha vida acadêmica.

Agradeço a todos os meus familiares que direta e indiretamente me apoiaram e me incentivaram nesta trajetória universitária.

Agradeço à minha namorada Letícia por todo apoio, carinho, amor, dedicação e incentivo ao longo da minha trajetória universitária.

Agradeço ao professor orientador Marcelo Maia por todo apoio, dedicação e comprometimento tornando este trabalho viável.

Agradeço aos meus colegas de turma Vitor Hugo e Lucas Moreira que participaram ativamente de grande parte da minha formação.

Agradeço aos professores Victor Sobreira e Fabiano Dorça pela disposição em participar da banca examinadora deste trabalho.

Agradeço ao corpo docente da Faculdade de Computação, Coordenação do Curso de Ciência da Computação e outras unidades que fizeram parte da minha formação.

"Nós só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a fazer."(Alan Turing)

Resumo

Pesquisadores na área de reparo automático trabalham constantemente desenvolvendo ferramentas dedicadas a corrigir *bugs* de sistemas automaticamente. As ferramentas de reparo, para serem adequadamente avaliadas e comparadas por meio de um estudo empírico, requerem benchmarks de bugs significativamente representativos dos bugs que ocorrem no mundo real. Um *benchmark* representativo é importante uma vez que as ferramentas podem ser mais efetivas em uma classe (ou tipo) específica de *bugs*, e portanto, *benchmarks* não representativos podem beneficiar ou prejudicar a avaliação de determinadas ferramentas de reparo. Este trabalho tem por objetivo caracterizar *benchmarks* de *bugs* conhecidos: o *Bugs.jar*, *Defects4J* e *Bears*; e avaliar se a ferramenta *ADD* (*Automatic Diff Dissection*) é adequada para caracterizar *benchmarks* de *bugs* automaticamente. Para alcançar os objetivos, primeiro será realizada a execução de *ADD* em todos os *bugs* dos *datasets* (caracterização de *Bugs.jar, Defects4J e Bears*); segundo, uma amostra estratificada dos *bugs* de *Bugs.jar, Defects4J e Bears* será coletada para a realização de uma análise manual com intuito de verificar os acertos e erros de *ADD* ao caracterizar os *bugs* (avaliação de *ADD*). Verificou-se que os *benchmarks* possuem características semelhantes em termos de ações e padrões de reparo, sendo modificações em chamada de métodos o grupo de ações mais evidente e a adição de blocos condicionais o grupo de padrões mais evidente. O *ADD* mostrou-se preparado para ser utilizado em novos *benchmarks*, acertando 99% dos itens analisados manualmente, correções podem melhorar seus resultados.

Palavras-chave: *benchmark de bug*, *Bugs.jar*, *Defects4J*, *Bears*, *reparo automático de software*.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A correção de *bugs* em sistemas é uma tarefa de manutenção de software que consome esforço significativo, o qual está associado ao processo de detectar o problema, diagnosticar a falha, identificar a causa raiz, e finalmente corrigir o *bug*. Ferramentas dedicadas à correção automática de *bugs* têm sido estudadas para mitigar este esforço (MONPERRUS, 2014).

No âmbito da correção automática de *bugs*, a avaliação empírica de ferramentas propostas é feita sobre um conjunto de *bugs* conhecidos (do inglês, *dataset of bugs*), que é normalmente chamado de *benchmark* de *bugs*. Em alguns casos, pesquisadores constroem seus próprios *benchmarks* de *bugs*, dedicando tempo de seu trabalho apenas para analisar e selecionar *bugs* conhecidos em sistemas reais para compor *benchmarks* de qualidade que permitam a avaliação de suas ferramentas (MADEIRAL et al., 2018).

Pesquisadores em reparo automático de programa precisam de *benchmarks* de *bugs*. A utilização de *benchmarks* de *bugs* existentes é uma opção razoável, pois evita a criação de novos *benchmarks*, considerada uma tarefa desafiadora. Pesquisas têm sido desenvolvidas focando na atividade de coletar *bugs* e seus reparos (chamados *patches*), com o intuito de disponibilizá-los a autores de ferramentas de reparo automático na forma de *benchmarks* de *bugs* (por exemplo, *Defects4J* Just, Jalali e Ernst (2014), *Bugs.jar* Saha et al. (2018), *Bears* Madeiral et al. (2019)). Estes *benchmarks* de *bugs* são criados por meio da mineração de *bugs* a partir de projetos do mundo real, como é o caso dos *benchmarks* *Defects4J* e *Bugs.jar*, onde seis e oito projetos são utilizados, respectivamente. Tais *benchmarks* normalmente são limitados a expor os *bugs* e suas origens no código-fonte de onde foram retirados, não necessariamente incluindo a caracterização dos mesmos. Em outras palavras, sabe-se quais são os *bugs* que existem nos *benchmarks*, mas em alguns casos não são conhecidas as características desses *bugs* (por exemplo, há *bugs* corrigidos usando apenas uma linha de código, enquanto outros *bugs* são corrigidos com uma instrução *if-then-else*) (SOBREIRA et al., 2018).

1.1 Problema

Este projeto tem dois problemas como alvo:

i) Alvo primário: somente *Defects4J* foi caracterizado, logo não se sabe a diferença entre *Defects4J* e outros *benchmarks* de *bugs*, como o *Bugs.jar* ou *Bears*.

ii) Alvo secundário: apurar a execução da ferramenta que será utilizada para atacar o problema específico i), *ADD (Automatic Diff Dissection)*, visando avaliar se os resultados

da execução da ferramenta são satisfatórios em diferentes *benchmarks*.

1.2 Objetivos

Os objetivos deste trabalho são: 1. avaliar o *ADD* em *benchmarks* de *bugs* já estudados (como é o caso de *Defects4j*) e *benchmarks* que ainda não foram estudados em relação a caracterização de seus *bugs* (como o *Bugs.jar* e o *Bears*); 2. verificar se a ferramenta *ADD* utilizada para caracterizar *benchmarks* pode ser utilizada em larga escala, isto é, em outros *benchmarks* de *bugs*, para este estudo serão considerados *Bears, Bugs.jar e Defects4J*.

1.3 Método

A metodologia empregada pode ser dividida em três partes:

1. **Coletar dados**. A primeira tarefa antes de executar a ferramenta *ADD* é obter os dados dos *benchmarks*. Para obter os arquivos necessários, código-fonte e *diff*, o *ADD* oferece um *software* em um repositório de hospedagem de *software* (servidor onde os arquivos de projetos são mantidos e podem ser compartilhados com público) onde estão todos os *bugs* a serem analisados. Logo, obter os arquivos foi possível através do *script*[1].

2. **Executar ADD**. A ferramenta *ADD* é executada somente em um *patch* específico por vez. Para cada execução, o *ADD* produz como saída um arquivo com a contagem das métricas de reparo encontradas no *patch*.

3. **Analisar resultados**. Por fim, é necessário avaliar o resultado obtido do *ADD*, isto é: caracterizar manualmente uma amostra estratificada dos *bugs* de cada um dos *benchmarks*. Com os dois resultados, resultado obtido da análise de uma amostra manual e os dados obtidos através da *ADD*, será possível caracterizar os *benchmarks* e também determinar a precisão de *ADD* em relação aos resultados obtidos por análise manual, semelhante ao que Sobreira et al. (2018) fez em *Defects4J*.

1.4 Resultados Esperados

Ao final do desenvolvimento deste projeto, espera-se:

i): A compreensão des *bugs* de três diferentes *benchmarks Bugs.jar*, *Defects4j* e *Bears*, isto é, conhecer os padrões nos *patches* usados para corrigir os *bugs*, o que permite caracterizar cada um dos *datasets*.

[1] https://goo.gl/xXYxju

ii): A confirmação de que a ferramenta *ADD* é adequada em relação a capacidade de detectar padrões de reparo automaticamente e que a mesma pode ser utilizada em outros *benchmarks* de *bugs*.

1.5 Justificativa

A simples existência de *benchmarks* de *bugs* não é suficiente aos pesquisadores em reparo automático de programa, pois é necessário que eles conheçam esses conjuntos de dados afim de ajuda-los a definir o *benchmarks* que usarão por exemplo se o pesquisador quiser desenvolver uma ferramente de reparo automático para corrigir um determinado segmento de problemas, ele precisaria avaliar todo um *benchmark* para verificar se aquele irá atende-lo em seus experimentos.

Diferentes *benchmarks* de *bugs* podem ser utilizados para o desenvolvimento e melhoria de ferramentas na tarefa de reparo automático de *bugs*, visto que esse tipo de ferramenta pode apresentar resultados diferentes dependendo das características do *benchmark* Durieux et al. (2019).

A ferramenta *ADD* pode ser usada exclusivamente a fim de obter as características específicas de *benchmarks* de *bugs* automaticamente. Usar *ADD*, além da economia de tempo (pois evita análise manual de dados), permite aos pesquisadores: i) avaliar suas ferramentas; ii) desenvolver ferramentas.

1.6 Organização do Trabalho

O restante deste trabalho está organizado como segue. O Capítulo 2 apresenta os conceitos básicos necessários para o entendimento deste trabalho e também são apresentados os trabalhos relacionados. O desenvolvimento é apresentado como documentação do software *ADD* no Capítulo 3 e o trabalho realizado de coleta dos dados e execução do *ADD* ficam no Capítulo 4. No Capítulo 5 são apresentados os resultados. O Capítulo 6 compreende as considerações finais.

2 Fundamentação Teórica

Neste capítulo são apresentados os conceitos básicos necessários para compreensão deste trabalho (seção 2.1) bem como os trabalhos relacionados (seção 2.2).

2.1 Conceitos Básicos

Nesta seção serão introduzidos termos e conceitos básicos das ferramentas e nomenclaturas utilizadas em torno do processo de automatização de correção de *bugs*.

2.1.1 Patch

Patch é uma mudança realizada no código-fonte de um programa, seja para adicionar uma nova funcionalidade ou para uma refatoração, por exemplo. Neste contexto, toda alteração em um arquivo de código pode ser refletida em um *patch*. A Figura 1 exemplifica um *patch* retirado de *Bugs.jar*, que corrige um *bug* no projeto *logging-log4j2*. Linhas em *verde*, marcadas com o símbolo "+", são as linhas adicionadas no programa pelo *patch*, e as vermelhas, marcadas com o símbolo "-", são as linhas removidas.

```
--- a/log4j-api/src/main/java/org/apache/logging/log4j/message/StringFormattedMessage.java
+++ b/log4j-api/src/main/java/org/apache/logging/log4j/message/StringFormattedMessage.java
@@ -133,7 +133,9 @@ public class StringFormattedMessage implements Message {
     stringArgs = new String[argArray.length];
     int i = 0;
     for (final Object obj : argArray) {
-        stringArgs[i] = obj.toString();
+        final String string = obj.toString();
+        stringArgs[i] = string;
+        out.writeUTF(string);
         ++i;
     }
 }
```

Figura 1 – *Patch* do *bug* 16ad8763 do projeto *logging-log4j2*.

Em reparo automático, *patches* podem ser utilizados como uma ferramenta para aprender como *bugs* são reparados por humanos, e consequentemente pesquisadores podem desenvolver ferramentas de reparo mais robustas. Por essa razão, *benchmarks* de *bugs* podem requerer o patch escrito por um humano para cada *bug* no *benchmark*.

2.1.2 Reparo automático de programa

Visto as dificuldades relacionadas ao trabalho manual de encontrar um *bug* e corrigi-lo, pesquisadores utilizam técnicas de diferentes áreas da ciência da computa-

ção para automatizar a localização e o reparo de *bugs* em programas, reparo este que é proposto na forma de *patch*.

A família mais conhecida de técnicas de reparo automático utiliza suítes de testes para confirmar se uma determinada ferramenta de reparo automático corrige um *bug* ou não. Suítes de testes são criadas junto ao código-fonte do programa e são compostas por casos de testes específicos do programa em desenvolvimento. Em cada caso de teste, uma funcionalidade do programa é testada a fim de verificar se as operações estão produzindo uma saída esperada para determinadas entradas. Caso contrário erros são reportados para os respectivos casos de teste. Trazendo para o contexto de reparo automático de programa, as suítes de teste expõem um *bug* quando pelo menos um caso de teste falha. O *patch* gerado por uma ferramenta de reparo corrige o *bug* se o suíte de teste inteiro passa quando executado na versão do programa com o *patch* produzido pela ferramenta de reparo.

Weimer et al. (2010) descrevem uma ferramenta de reparo automático chamada *GenProg* (do inglês, *Genetic Program Repair*). Através da programação genética (uma busca estocástica inspirada na biologia evolutiva que colabora em programas de computador para determinadas tarefas), inspirada no *crossover* da biologia celular, a ferramenta *GenProg* computa variações do programa a ser reparado, até que se encontre uma variante do programa que passe em todos os casos de testes, denominado *reparo primário* em *GenProg*. Como esse reparo é o final de várias operações de um algoritmo genético, há modificações que não influenciam o *bug* em questão, logo o *reparo final* é uma minimização do *reparo primário*, removendo as modificações que não afetam os casos de testes e o fluxo do reparo. Com isso é possível determinar um *diff* entre o programa inicial e o programa cujo código repara o *bug* que falha em casos de testes a fim de produzir um *patch* final. Conclui-se que *GenProg* possui uma taxa de 77% em reparos, os experimentos foram realizados utilizando 16 programas escritos em C.

Xuan et al. (2017) desenvolveram uma ferramenta de reparo automático de *bugs* denominada *Nopol*. Os *bugs* que eles propuseram corrigir só ocorrem envolvendo estruturas condicionais (*if-then-else*). Eles afirmam que estruturas condicionais é o elemento mais propenso a erro em Java e verificaram em um conjunto de dados que havia muitos *commits* (em versionamento de código-fonte, *commit* significa atualizar o repositório com a última versão de código) que só atualizam este tipo de estrutura. Casos de estudos foram apresentados para a ferramenta proposta através de *benchmark* sintetizado de uma amostra de *bugs* produzida utilizando programas que também foram utilizados em *Bugs.jar* e *Defects4J* a fim de avaliar a ferramenta proposta.

Nopol, através de uma suíte de testes do programa defeituoso, procura por estruturas condicionais ou blocos de códigos onde supostamente há uma necessidade de uma precondição (e.g., chamada de método, estrutura de *loop*, assinatura de variável). Para encontrar os *bugs*, *Nopol* utiliza uma técnica chamada, em inglês, *Angelic Fix Localiza-*

tion, como trata-se de estruturas condicionais onde é necessário uma expressão booleano de entrada e o resultado é um valor booleano (verdadeiro ou falso), esta técnica altera os valores de expressões booleanas até que os casos de testes que estão falhando parem de falhar. Quando um teste que estava falhando para de falhar, *Nopol* então realiza uma coleta de informações das variáveis e os tipos primitivos de dados no contexto do candidato a *bug* (XUAN et al., 2017).

Por fim, é necessário sintetizar um *patch* a fim de solucionar o *bug* em questão, *Nopol* realiza este procedimento transformando os dados coletados no contexto do local onde possivelmente está o *bug* em uma fórmula *SMT*, do inglês (*Satisfiability Modulo Theory*). O *patch* final a ser produzido precisa satisfazer a *SMT*. Para satisfazer é necessário que o *patch* sintetizado não altere o fluxo do programa em questão e o caso de teste no qual ele pertence seja executado com sucesso. Ao final deste processo, a suíte de teste é executada para verificar se nenhum caso de teste falhou.

2.1.3 Benchmarks de bugs

Benchmarks de *bugs* são criados para apoiar pesquisadores que realizam estudos empíricos sobre *bugs* de *software* (e.g. reparo automático de *bug*). Os *benchmarks* de *bugs* são criados com a finalidade de reproduzir uma amostra de *bugs* presentes em projetos de *software* do mundo real, para que pesquisadores possam criar estudos reproduzíveis, além de reduzir os custos da manutenção de *software* (JUST; JALALI; ERNST, 2014; SAHA et al., 2018).

Just, Jalali e Ernst (2014) criaram um *benchmark* contendo 395 *bugs*, chamado *Defects4J*. Os dados foram coletados a partir de 6 projetos Java, grandes, populares, e *open-source*, isto é, projetos que possuem o código-fonte disponível para que a comunidade de desenvolvedores possa contribuir com os mesmos e utilizá-los seguindo licenças de distribuição e uso. A Tabela 1 apresenta os projetos utilizados em *Defects4J*, bem como suas características quanto ao propósito e o número de *bugs*.

Tabela 1 – Projetos utilizados em Defects4J.

Projeto	Propósito	# *Bugs*
Closure Compiler	compilador JavaScript	133
Commons Lang	biblioteca extra para o pacote java.lang.*	65
Commons Math	biblioteca matemática e estatística	106
JFreeChart	biblioteca para trabalhar com gráficos em java	26
Joda-Time	biblioteca para trabalhar com data em java	27
Mockito	Framework de testes	38
Total		**395**

Saha et al. (2018) construíram um *benchmark* de *bugs* chamado *Bugs.jar*, esse

é formado por 1.158 *bugs*. Semelhante ao *Defects4J*, *Bugs.jar* também contém *bugs* de projetos *open-source*, inclusive o objetivo principal na construção de *Bugs.jar* foi aumentar a quantidade e diversidade de *bugs* coletados em relação ao *Defects4J*. Os projetos utilizados em *Bugs.jar* são apresentados na Tabela 2, onde o propósito de cada projeto também é apresentando, e por fim, na última coluna, a quantidade de *bugs* extraídos de cada projeto, que são contidos em *Bugs.jar*.

Tabela 2 – Projetos utilizados em Bugs.jar.

Projeto	Propósito	# *Bugs*
Accumulo	classificador, armazenamento chave-valor	98
Camel	Motor de rotas e intermédio	147
Commons Math	Biblioteca matemática e estatística	147
Flink	Motor de *streaming*	70
Jackrabbit Oak	Sistema gerenciador de conteúdo	278
Log4J2	*Framework* de log	81
Maven	Gerenciador de projetos	48
Wicket	Aplicativo Web para servidor	289
Total		**1.158**

Madeiral et al. (2019) propôs um *benchmark* semelhante ao *Bugs.jar* e *Defects4J*, porém sua diferença está na forma como o mesmo encontra *bugs*. *Bears*, contém um sistema que permite de forma automática procurar *bugs* e a forma como os mesmos foram corrigidos, isso através do uso da integração contínua (*CI* do inglês *Continuous Integration*). Baseado no estado da integração contínua (*passed*, *error* ou *failed*) é possível mapear os estados do software onde havia um *bug* (*error* ou *failed*) e quando o *bug* foi corrigido: *passed*.

É fundamental para a existência do *Bears*, dois sistemas: o *GitHub*[1] (sistema de repositório de código online que utilizam *git*) e *Travis CI*[2] (sistema de *build* automático para repositórios de código online, como o *GitHub*). O *Travis* realiza o papel de retirar o código-fonte do *GitHub* (ou outro repositório de código online) e realizar o processo de *build*, isto é: compilar e executar os testes unitários do projeto. O processo de *build* do *Travis* pode desempenhar outros papéis além de compilar e executar os testes unitários, porém estas duas tarefas são muito importantes: pois detectam a cobertura dos testes (medida que define quanto do código foi testado automaticamente) e se o projeto passou em todos os casos de testes. Baseado em todos os processos de *build* em um projeto, *Bears* procura por um par de *build*, onde o primeiro contém uma falha e o segundo um possível *patch* de correção Madeiral et al. (2019).

[1] https://github.com
[2] https://travis-ci.org/

2.1.4 UML *Unified Modeling Language*

A UML é uma linguagem para especificar, visualizar, construir e documentar sistemas de *software*. A linguagem foi criada pela OMG (*Object Management Group*)[3]. Ela compreende boas práticas de engenharia bem sucedidas na modelagem de sistemas grandes e complexos. Essa modelagem ocorre através de Diagramas Gráficos.

A escolha de quais diagramas serão utilizados depende de como o problema a ser solucionado será atacado e de qual forma será modelado. A UML define alguns diagramas, que podem ser divididos em dois grandes grupos: Diagramas de Comportamento e Diagramas de Estrutura.

Os diagramas de comportamento, expressam os comportamentos dos objetos em um sistema, descritos por uma série de alterações que o sistema sofre ao decorrer do tempo. Os principais diagramas utilizados para descrever esses comportamentos e mudanças são: diagrama de casos de uso, diagrama de atividades, diagrama de estados e diagrama de comunicação.

Os diagramas de estrutura são utilizados para mostrar estruturas estáticas do sistema em diferentes níveis de implementação e também como estas partes estão relacionadas. Os principais diagramas de estrutura são: diagrama de classes, diagrama de pacotes, diagrama de componentes e diagrama de implantação.

2.2 Trabalhos Relacionados

Nesta seção serão apresentados dois trabalhos: subseção 2.2.1 apresenta a caracterização de um *benchmark* de *bugs*, o *Defects4J*, bem como a definição de 9 grupos de características; subseção 2.2.2 apresenta a ferramenta *ADD*, tem objetivo de caracterizar automaticamente *patches* e cujo experimentos foram realizados em *Defects4J*.

Sobreira et al. (2018) trouxe a público informações qualitativas (e.g. padrões de reparo) e também as informações quantitativas (e.g. métricas sobre os *patches*) sobre *Defects4J* e o trabalho de Madeiral et al. (2018) introduziu o *ADD* que foi executado somente no *Defects4J*.

2.2.1 Caracterização de benchmarks de bugs

Sobreira et al. (2018) propõe uma caracterização com base em um estudo conduzido sobre um *benchmark* de *bugs* conhecido, o *Defects4J*. Tal caracterização foi realizada manualmente comparando os *patches* de código-fonte da solução de cada *bug* de *Defects4J*. A caracterização proposta por Sobreira et al. (2018) e colegas resultou nos seguintes dados:

[3] https://www.omg.org/spec/UML/2.5.1/PDF

1. Tamanho do *patch*: contagem das linhas que são adicionadas, removidas ou modificadas em um *patch*;

2. Espalhamento do *patch*: contagem do espalhamento do *patch* isto é, um *patch* pode ser composto por uma sequência contínua de códig (ou seja, um pedaço ou Chunk), ou pode ser composto de vários pedaços separados por no mínimo uma ou mais linhas;

3. Ações de Reparo: são os blocos construídos para o *patch*, isto pode ser uma chamada de método, uma estrutura condicional, atribuição de valor à uma variável, laço, definição de método, instância de objetos, exceções, retorno de método e ações relacionadas a declaração de variáveis;

4. Padrões de Reparo: com o conhecimento e análise obtido através das ações de reparo, foi observado uma recorrência de estruturas nos *patches*, com uma análise temática, obteve-se os seguintes padrões: bloco condicional, correção de expressões, *wraps-with*, apenas uma linha, referência incorreta, falta de checar um *null*, copiar/colar, alteração de constante e movimentação de código.

2.2.2 Ferramenta para caracterização de benchmarks de bugs

Madeiral et al. (2018) apresentou uma ferramenta para automatizar a busca de propriedades de *patches* apresentadas em Sobreira et al. (2018). A ferramenta desenvolvida chama-se *ADD* e é objeto de estudo nesta monografia.

ADD realiza uma análise da estrutura sintática da diferença entre dois arquivos de código, o código na qual contém o *bug* e o código que contém o *patch* que resolve o *bug* e tenta descrever em termos de características (e.g. *Bloco Condicional* (*Conditional Block*), *única linha* (*Single Line*)) o *patch* usado para solucionar o *bug* em questão. A análise sintática ocorre através de uma representação do código chamada de *AST* (do inglês Abstract Syntax Tree), com a *AST* de um arquivo de código é possível lidar programaticamente com cada parte do código (e.g. estruturas condicionais, declaração de variáveis, estruturas de repetição), logo combinando as características apresentadas em Sobreira et al. (2018) e as *AST* de *bug* e *patches* é possível determinar automaticamente os padrões de reparo (MADEIRAL et al., 2018).

Madeiral et al. (2018) utilizaram o *Defects4J* para experimento em relação a precisão da *ADD*. O experimento foi realizado utilizando padrões de reparo reportados por humanos. Foi notado que a *ADD* tem uma precisão alta comparada aos resultados obtidos por humanos. Para o grupo mais recorrente, *Bloco Condicional* (*Conditional Block*), a *ADD* obteve em média (há quatro padrões diferentes para *Bloco Condicional*) 95.6% de precisão, para padrões menos recorrentes como é o caso do *Única Linha* (*Single Line*),

ADD obteve 100% de precisão. Em alguns padrões específicos a *ADD* não se comportou muito bem, gerando resultados falso-positivos.

3 Documentação Automatic Diff Dissection

Este capítulo visa apresentar uma documentação do código-fonte do *Automatic Diff Dissection*[1] através da UML *(Unified Modeling Language)* apresentando informações relevantes sobre suas características, relações entre os componentes, organização do projeto e como pesquisadores podem modificar o projeto com o objetivo de acrescentar novas características de reparo (ações ou padrões).

Documentar um sistema é prover informações precisas, uma descrição detalhada sobre o mesmo, uma forma de comunicação entre os desenvolvedores. O trabalho de documentar pode não ser um resultado colhido apenas por desenvolvedores, pode ser também utilizado para construir manuais de produto para pessoas que não são desenvolvedores. A documentação pode ser apresentada em diversos formatos, tais como: texto, modelos gráficos (UML) e sítios na internet. Alguns pesquisadores acreditam que o código-fonte pode ser considerado uma documentação do sistema (ZHI et al., 2015).

O sistema em estudo foi construído utilizando a linguagem de programação Java, cujo paradigma de programação utilizado para a linguagem é o orientado a objetos. A orientação a objetos permite construir sistemas robustos criando abstrações do mundo real utilizando conceitos bem definidos como classe, objeto, herança, polimorfismo e encapsulamento.

Utilizando o Diagrama de Classes junto ao poder da orientação a objetos é possível definir as classes para manipular e processar o problema a ser resolvido. Essa definição inclui atributos, métodos, interfaces e classes abstratas. Além disso, é possível conhecer as relações de herança entre as classes definidas. O grande poder de desenvolver um diagrama de classes antes de digitar o código-fonte é a facilidade que se tem em modificar o diagrama até que ele atenda corretamente os requisitos para resolver o problema em questão, *atender corretamente* os requisitos não é simplesmente resolver o problema, mas sim resolvê-lo de uma forma organizada e simples, utilizando os conceitos de orientação a objetos, de tal forma que as manutenções no código e futuros contribuintes sejam capazes de ler, entender e contribuir com o projeto.

O projeto *ADD* contém ao todo 37 tipos, incluindo classes, classes abstratas e interfaces, além das classes comuns. O projeto contempla testes unitários a fim de aferir a corretude do código escrito.

A Figura 2 ilustra as classes utilizadas para encontrar os padrões de ações utilizados para reparar um *bug*, denominados *Repair Actions*. Uma classe muito importante definida nessa imagem é a classe **Feature**. Esta classe é responsável por atualizar informações

[1] https://github.com/lascam-UFU/automatic-diff-dissection

sobre as *features*, sejam elas *Repair Actions* ou *Repair Patterns*, encontradas durante a varredura de um *bug* em análise. Essa atualização é na grande parte incrementar o número que conta quantas *features* foram encontradas. A classe também é responsável por exportar os dados após a análise realizada, permitindo exportar os dados tanto em formato *CSV* como também em formato *JSON*. O único atributo que a classe **Feature** possui, chamado *config* do tipo *Config*, retém as informações que o usuário informou ao executar o *ADD*.

Em termos de configuração de inicialização, o usuário deve informar os argumentos especificados na Tabela 3:

Tabela 3 – Parâmetros de entrada para execução do ADD

Propriedade	Valores Esperados
launcherMode	REPAIR_PATTERNS; REPAIR_ACTIONS; METRICS; ALL
bugId	Essa informações somente é utilizada para apresentar os resultados
buggySourceDirectory	Diretório do código onde está o bug
diff	Caminho onde está o arquivo diff
output	Diretório onde os dados serão exportados

Para que pesquisadores possam incrementar o funcionamento do *ADD* com novas funcionalidades é necessário o entendimento da biblioteca *Spoon*. *Spoon* possui 5 funcionalidades básicas: 1. representação de código Java através de árvore sintática abstrata; 2. *API* de interseção para modificar e gerar código Java; 3. tipos genéricos para análise de código estática; 4. integração e processamento nativo de anotações Java; 5. um motor de checagem estática Pawlak et al. (2015).

Falleri et al. (2014) o algoritmo GumTree é utilizado em *Spoon*, o que possibilita o *ADD* interpretar cada trecho que foi inserido, deletado ou modificado em um *patch* e é desta maneira que *ADD* detecta as ações e padrões de reparo e também as métricas: varrendo todas as mudanças de um *patch*.

A ferramenta *Spoon* expõe ferramentas que permitem ao *ADD* analisar profundamente cada elemento de trechos de códigos através de uma representação exclusiva que identifica os objetos, as classes, os tipos, as chamadas de métodos, as estruturas condicionais e outras operações que podem ser expressadas na linguagem Java.

O diagrama representado pela Figura 2 demonstra a organização das classes que permitem ao *ADD* contabilizar e identificar ações de reparo.

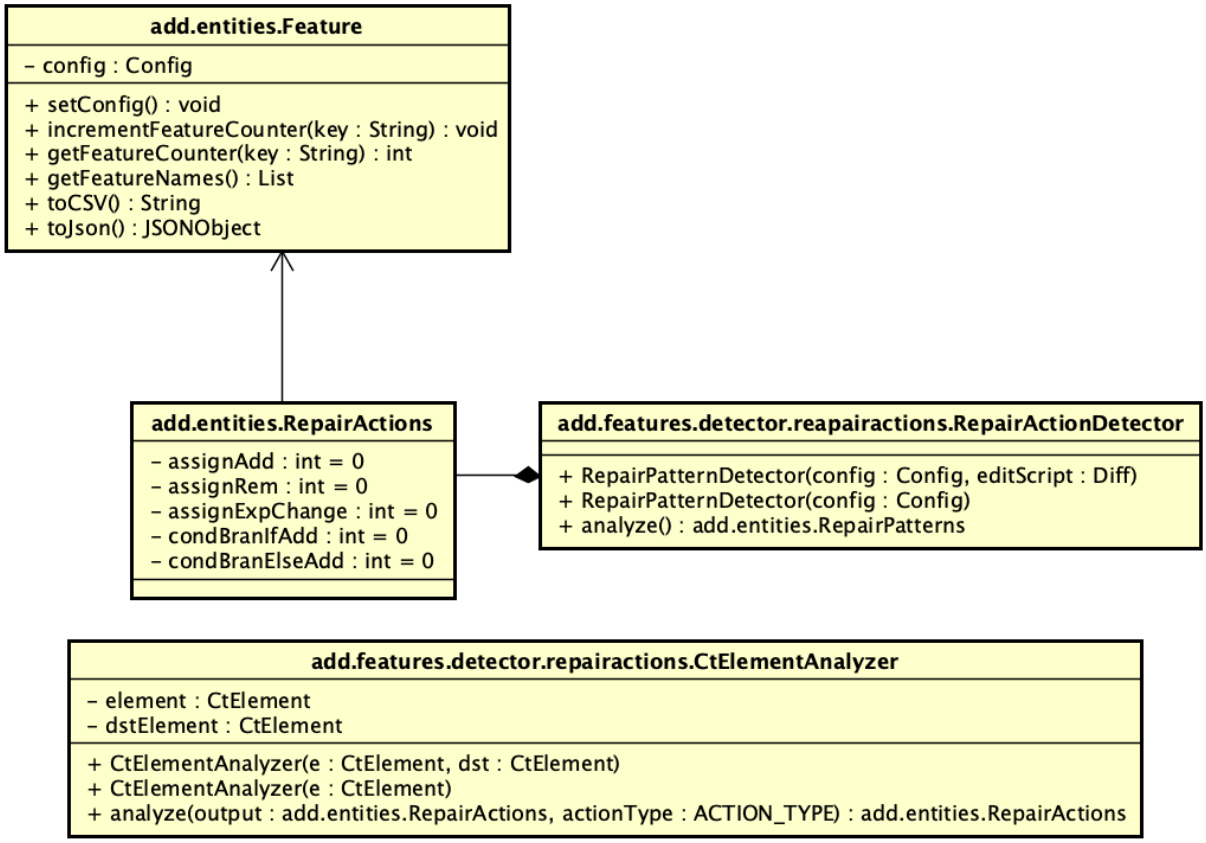

Figura 2 – Diagrama de classes envolvidos para detecção de *RepairActions*

A classe RepairActions retém as informações que serão utilizadas para contabilizar quantas ações foram encontradas em cada trecho de código do *patch*, isso explica o fato dessa classe conter vários atributos (a Figura 2 omite grande parte deles). Todos os atributos desta classe são do tipo inteiro e o que faz de um atributo dentro desta classe uma ação de reparo que será reportada no término da execução é a utilização de *Java Annotations*: os atributos são anotados por @FeatureAnnotation, isto é, são sinalizados para que sejam identificados e manipulados posteriormente sem a necessidade de utilizar encapsulamento. A Figura 3 exemplifica como a ação de reparo *Assignment Addition* é declarada.

```
@FeatureAnnotation(key = "assignAdd", name = "Assignment addition")
private int assignAdd = 0;
```

Figura 3 – Exemplo da Anotação utilizada em atributos de ações e padrões de reparo do *ADD*

A varredura do *Spoon* ocorre através da classe CtElementAnalyser onde também está implementada a lógica para detectar cada uma das ações de reparo. Esta lógica

é executada no momento em que o método ANALYZE é executado, logo para que um pesquisador possa estender as ações de reparo, os principais passos para isso são:

1. Adicionar um atributo do tipo *int* na classe REPAIRACTIONS;
2. Anotar o novo atributo adicionado semelhante ao exemplo apresentado na Figura 3;
3. Utilizar o *Spoon* dentro da classe CTELEMENTANALYZER especificando qual tipo de operação em Java (por exemplo, estrutura condicional, chamada de método, instância de objetos) e a lógica necessária afim de identificar a nova ação de reparo;
4. Quando identificada a ação de reparo deverá ser contabilizada utilizando o atributo definido no passo 1;
5. Ao término da execução do *ADD* todas as ações encontradas serão automaticamente exportadas em um arquivo no formato *JSON*.

Em termos de padrões de reparo a implementação do *ADD* está segregada por classes onde cada classe representa um padrão, logo cada classe implementa a lógica para identificar e contabilizar o padrão desejado. Todo padrão implementado no *ADD* é uma extensão da classe ABSTRACTPATTERNDETECTOR que contém o método DETECT que é usado para executar a lógica do padrão.

A Figura 4 ilustra a implementação do padrão WRONGREFERENCEDECTECTOR (outros padrões foram omitidos). A classe REPAIRPATTERNS é semelhante a classe REPAIRACTIONS em termos de atributos e também em termos da anotação utilizada para identificar e manusear cada um dos padrões de reparo.

A classe REPAIRACTIONDETECTOR tem por objetivo inicializar todas as classes de padrão de reparo e executar o método ANALYZE de cada uma delas.

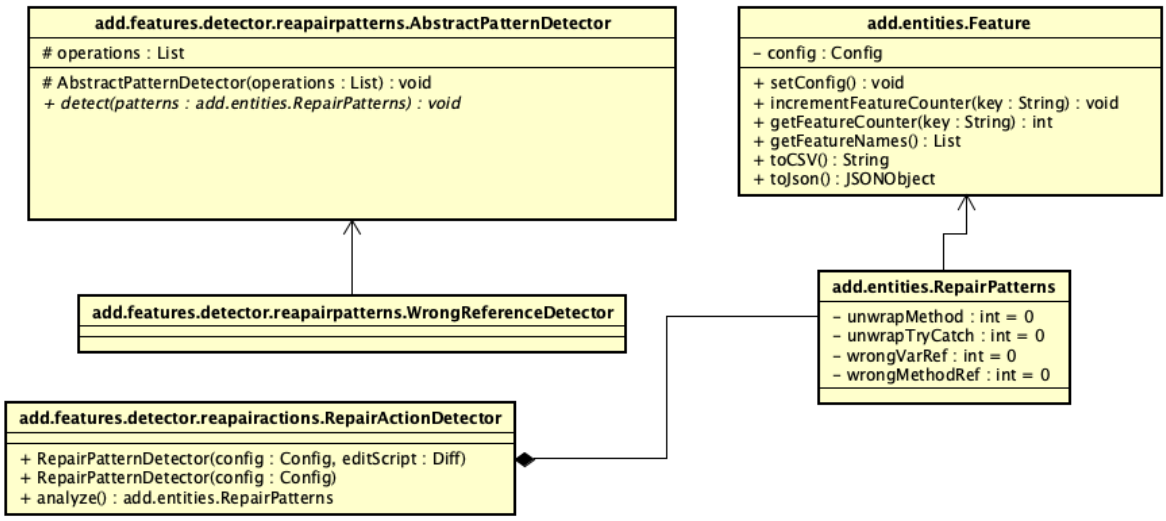

Figura 4 – Diagrama de classes das classes responsáveis pelos padrões de reparo no *ADD*

Para implementar um novo padrão de reparo é necessário:

1. Adicionar um atributo do tipo *int* na classe RepairPatterns;
2. Anotar o novo atributo adicionado semelhante ao exemplo apresentado na Figura 3;
3. Criar uma classe que estenda a classe AbstractPatternDetector;
4. Implementar a lógica para identificar e contabilizar o padrão de reparo no método detect da nova classe criada;
5. Na classe RepairActionDetector é necessário criar uma instância do novo padrão e configura-lo para que possa ser executado junto aos demais padrões já existentes;
6. Ao término da execução do *ADD* todas as ações encontradas serão automaticamente exportadas em um arquivo no formato *JSON*.

4 Desenho do Estudo

Este capítulo apresenta os métodos utilizados neste estudo de tal forma a responder às seguintes perguntas de pesquisa:

- PP#1: Qual é a acurácia do *ADD*? A acurácia será definida via análise manual de uma amostra de cada um dos *benchmarks*, onde os resultados do *ADD* serão comparados com a análise manual.
- PP#2: Quais são os padrões e ações de reparos que caracterizam cada um dos *benchmarks*? Serão listados o Top 5 de ações e padrões de reparo de cada um dos *benchmarks* visando destacar as características expressivas de cada um dos *benchmarks*.

A seguir serão apresentados os *benchmarks* a serem utilizados neste estudo, depois o processo de execução da ferramenta *ADD* nesses *benchmarks*, e finalmente os detalhes experimentais desta investigação.

4.1 Dataset

A primeira etapa do processo para executar o *ADD* é coletar os dados dos *benchmarks*, isto é, os *bugs* endereçados por tais *datasets*.

Os *benchmarks* envolvidos neste trabalho diferenciam-se uns dos outros na forma como os dados são disponibilizados e também na fonte de onde vieram as informações, isto é, os projetos que cada um dos respectivos autores analisaram e reuniram as devidas informações. Por fim é importante ressaltar que cada um dos *benchmarks* abrangem projetos OpenSource amplamente utilizados na indústria como é o caso do *Maven*[1], *FasterXML*[2], *Apache Commons Math*[3] e outros Saha et al. (2018), Just, Jalali e Ernst (2014) e Madeiral et al. (2019).

Esses *datasets* possuem um conjunto de arquivos que representam os *bugs*, dentre eles os dois que são utilizados pelo *ADD* são: versão do código-fonte com o *bug* e o *diff* entre a versão do código com o *bug* e a versão do código que implementa a correção.

Estes são dois arquivos de importância no *benchmark*:

[1] https://github.com/apache/maven
[2] https://github.com/FasterXML
[3] https://github.com/apache/commons-math

- **Arquivo do código-fonte** que foram utilizados no experimento representam o estado anterior antes do *patch* de correção ser aplicado em cada um deles. Esses arquivos representam classes Java, logo todos os *bugs* analisados pelo *ADD*, e por consequência os analisados manualmente são de projetos escritos em Java.
- ***Patch* do código-fonte** apresenta a diferença entre a versão com código que possui o *bug* e a versão código-fonte após a correção do *bug*, conforme apresentado abaixo na Figura 7.

No estudo de Madeiral et al. (2018), os dados dos *benchmarks* estudados foram coletados e reunidos em um repositório público e neste mesmo trabalho foi disponibilizado o código-fonte do *ADD* e a documentação para executar o mesmo utilizando os *benchmarks*.

Há uma certa diferença para identificar as entradas de cada um dos *benchmarks*. Em *Bears* todas as entradas seguem o seguinte padrão de identificação: **Bears_A** onde é um número inteiro natural maior que zero, sendo assim são exemplos de elementos pertencentes ao *Bears*: **Bears_1**, **Bears_2**, etc.

Em *Defects4J* (JUST; JALALI; ERNST, 2014), cada entrada do *benchmark* recebe uma identificação parecida com *Bears*, porém a diferença está no prefixo: cada entrada em *Defects4J* é identificado pela inicial do projeto e um número natural maior que zero, por exemplo: **Chart_1**, **Clousure_40**, **Lang_22**, etc.

Por fim, em *Bugs.jar* (SAHA et al., 2018), as entradas do *benchmark* são identificadas pela inicial do projeto e o código *hash* do *commit*. O *Hash* de um *commit* em um sistema de versionamento, é uma sequência originada a partir de um Algoritmo *Hash*, como o *SHA-256*, que recebe como entrada os dados do *commit*, obtendo-se então uma sequência *Hash*. Os 8 primeiros caracteres já são suficientes para identificar um *commit*, logo cada elemento de *Bugs.jar* é agrupado pelo projeto e diferenciação entre itens do mesmo projeto ocorre através dos 8 primeiros caracteres do *Hash* do *commit*, por exemplo: **Accumulo@47c64d9a**, **Commons-math@4c4b3e2e**, etc.

4.2 Execução

O processo de execução do *ADD* nos *benchmarks* citados ocorre através de três passos:

- Obtenção dos *benchmarks*;
- Obtenção do *ADD*;
- Execução do *ADD* nos *benchmarks* obtidos.

Madeiral et al. (2019) tornaram a ferramenta *ADD* disponível via repositório público de código, o GitHub[4]. Neste repositório encontra-se todo o código-fonte do projeto que foi escrito em Java, instruções de utilização e instruções para obtenção dos *benchmarks*.

A fim de auxiliar na execução foi desenvolvido um *script* para paralelizar o processo manual. Nos experimentos o *ADD* foi executado a fim de obter todas as *features* de cada *bug* analisado: *repair actions*, *repair patterns* e as métricas do reparo efetuado.

Ao todo foram 1.804 execuções do *ADD* ao longo dos três *benchmarks* e dessas execuções apenas 17 falharam, logo foram 1.787 execuções bem sucedidas.


```
{
    "repairActions": {
        "condBranElseAdd": 0,
        "mcAdd": 1,
        "loopAdd": 0,
        "mdModChange": 0,
        "mcRem": 0
        ....
    },
    "bugId": "math@math_10",
    "metrics": {
        "removedLinesCodeOnly": 0,
        "patchSizeCodeOnly": 1,
        "patchSizeAllLines": 1
        ....
    },
    "repairPatterns": {
        "wrapsLoop": 0,
        "wrapsTryCatch": 0,
        "wrapsIfElse": 0,
        "wrongMethodRef": 0,
        "constChange": 0
        ....
    }
}
```


Figura 5 – Exemplo de resultado do *ADD*

Os resultados do *ADD* são apresentados através de um arquivo exportado em formato *JSON* (*JavaScript Object Notation*) que é dividido em três grupos: *repair actions*, *repair patterns* e *metrics*. Cada grupo do arquivo apresenta todos os itens que a ele pertence e a cada um deles é associado um número que representa a quantidade de ocorrências daquela *feature* no *bug* analisado, conforme apresentado na Figura 5 (as reticências indicam que há outras informações).

4.3 Experimento

O experimento foi dividido em duas etapas distintas: a primeira etapa refere-se a um estudo qualitativo enquanto a segunda etapa refere-se a estudo quantitativo.

4.3.1 Estudo Qualitativo

Uma amostra de cada um dos *benchmarks* foi analisada manualmente, isto é, determinar os padrões e ações de reparo de um *patch*. A análise manual aconteceu através

[4] https://github.com/lascam-UFU/automatic-diff-dissection

de um estudo empírico, primeiro anotou-se os resultados manuais e depois os resultados foram comparados com os dados obtidos do *ADD*.

Analisar manualmente o resultado do *ADD* responde a PP#1, onde é possível avaliar os acertos e erros do *ADD* em uma perspectiva qualitativa. Os levantamentos produzidos nesta etapa são baseados em testes funcionais da aplicação, ou seja, teste de caixa de preta. Um teste de caixa preta ou teste funcional, é o processo de observar um sistema sob testes, os casos de testes são determinados a partir das especificações do sistema e o foco principal está no comportamento externo da aplicação Nidhra e Dondeti (2012).

```
1   --- a/org/apache/commons/math3/analysis/differentiation/DSCompiler.java
2   +++ b/org/apache/commons/math3/analysis/differentiation/DSCompiler.java
3   @@ -1416,6 +1416,7 @@ public void atan2(final double[] y, final int yOffset,
4           }
5
6           // fix value to take special cases (+0/+0, +0/-0, -0/+0, -0/-0, +/-infinity) correctly
7   +       result[resultOffset] = FastMath.atan2(y[yOffset], x[xOffset]);
8
9       }
10
11
```

Figura 6 – Caso de uso para exemplificar estudo qualitativo

A primeira etapa da análise manual consiste em verificar os padrões e ações de reparo conforme as definições em Sobreira et al. (2018). Na Figura 6, por exemplo, mostra que uma linha foi adicionada, a linha 7 e é observada adição de uma chamada de método e também uma atribuição, em termos de *ações de reparo*. Já em termos de *padrões de reparo* pode-se concluir que há uma única ocorrência do padrão *linha única*, isto é, a modificação ocorreu apenas em uma linha.

4.3.2 Estudo Quantitativo

Com os dados referentes as inúmeras execuções do *ADD*, torna-se viável consolidar os resultados em termos de valores estatísticos, bem como responder a PP#2: atribuir a cada um dos *datasets* os padrões e ações de reparos característicos.

Os resultados e definições apresentados nesta etapa ocorrem via estudo estatístico caracterizando então cada um dos *benchmarks* em termos de:

- Padrões de reparo mais recorrentes;
- Ações de reparo mais recorrentes;

5 Resultados

Este capítulo apresenta os resultados obtidos a partir dos experimentos destacados no Capítulo 4.

5.1 Bears

Foram manualmente analisados 20 bugs aleatórios do *benchmark Bears* Madeiral et al. (2019). Após efetivação da análise manual, um dos bugs analisados apresentou uma característica que ainda não é explorada pelo *ADD*: *Java Annotations*. O *Patch* em questão havia modificações apenas em anotações, como mostra a Figura 7, logo no resultado do *ADD* não consta nenhum padrão ou ação de reparo.

```
1  --- a/src/main/java/edu/harvard/h2ms/domain/core/User.java
2  +++ b/src/main/java/edu/harvard/h2ms/domain/core/User.java
3  @@ -36,7 +36,7 @@ public class User implements UserDetails {
4     @Column(name = "ID")
5     private Long id;
6
7  -  @NotNull @Column @JsonIgnore private String firstName;
8  +  @NotNull @Column private String firstName;
9
10    @Column private String middleName;
11
12 @@ -50,7 +50,7 @@ public class User implements UserDetails {
13
14    @NotNull @Column private String type;
15
16 -  @NotNull @Column private String password;
17 +  @NotNull @JsonIgnore @Column private String password;
18
19    @ManyToMany
20    @JoinTable(
21
```

Figura 7 – *Patch* com modificações em anotações

Em 4 *patches*, o que representa 20% da amostra observada, a análise manual identificou ações de reparo que o *ADD* não identificou, isto é, além das ações de reparo reportadas pelo *ADD*, a análise manual sugere mais algumas ações. As ações não registradas pelo *ADD* estão representadas na Tabela 4 junto ao Projeto analisado e ao seu identificador.

Tabela 4 – Ações de reparo que o ADD não identificou em Bears

Ação de Reparo	Projeto	Identificador
Method Call Remove	fasterxml-jackson-databind	Bears_16
Return Expression Modification	fasterxml-jackson-databind	Bears_24
Return Expression Modification	aicis_fresco	Bears_145
Var Replace var	fasterxml-jackson-databind	Bears_24
Loop Condition Change	inria-spoon	Bears_42

Ao todo em *Bears* foram analisados 251 *bugs*, deles foram retirados uma soma total de 1.222 padrões de reparos e 4.305 ações de reparo, o que sugere que há aproximadamente em média 5 padrões de reparo e 17 ação de reparo em cada projeto.

Quanto às ações de reparo, a Tabela 5 apresenta o Top 5.

Tabela 5 – Ações de reparo que são recorrentes em Bears

Ação de reparo	**Ocorrências**
Method Call Add	912
Method Call Parameter Value Change	447
Method Call Removed	324
Assign Add	280
Conditional Branch Add	277

Os padrões de reparos recorrentes do *benchmark* estão apresentados na Tabela 6.

Tabela 6 – Padrões de reparos que são recorrentes em Bears

Padrão de reparo	**Ocorrências**
Conditional Block Others Add	151
Copy Paste	125
Conditional Block Return Add	97
Constant Change	90
Wrong Method Reference	87

Os resultados apresentados nas Tabelas 5 e 6 mostram que 39% das ações de reparo realizadas nos projetos em *Bears* foram alterações relacionadas a chamada de métodos: adição de chamada de método, alteração em valores de parâmetros em chamada de métodos e remoção de métodos. Em termos de padrões de reparo, o *dataset Bears* apresentou todos padrões de reparo.

5.2 Defects4J

As análises realizadas no *benchmark Defects4J* Just, Jalali e Ernst (2014) somam 38 análises manuais e a execução completa do *ADD* em todos os 395 *patches* do *dataset*. Diferentemente de *Bears*, a análise manual junto aos resultados do *ADD* em *Defects4J* apresentou dois novos tipos de falhas em relação às falhas apresentadas na Seção 5.1: detecção de ações e padrões que a análise manual determinou que não existem e detecção de ações e padrões incorretos, isto é, o padrão ou ação está relacionada com o que foi reportado pelo *ADD* porém poderiam ter sido classificados de outra maneira.

Tabela 7 – Ações de reparos que o ADD não identificou em Defects4J

Ação de Reparo	**Projeto**	**Identificador**
Method Call Parameter Value Change	Closure	Closure_9
Assign Expression Change	Lang	Lang_33
Loop Condition Change	Math	Math_41
Condition Expression Modification	Closure	Closure_3

O último item da Tabela 7 aponta que no projeto Closure, o *patch* para o bug Closure_3, apresenta ao todo 10 ações de reparo identificadas pelo *ADD*. No ponto de vista de análise manual, há duas falhas: a primeira é que na identificação da ação Method Call Paramenter Value Change, o ADD aponta 23 ocorrências do mesmo, porém elas não ocorrem para o *patch*.

Em relação a *Condition Expression Modification*, também de *Closure_3*, o autor do trabalho aponta a existência desta ação, pois o método chamado dentro da condição, Figura 8, de uma estrutura condicional, é alterado o que reflete diretamente o resultado daquela instrução que por conseguinte refletirá, em tempo de execução do código, o resultado da condição.

```
6       for (Candidate c : candidates) {
7   -     if (c.canInline()) {
8   +     if (c.canInline(t.getScope())) {
9             c.inlineVariable();
10
```

Figura 8 – Closures_3: Detalhes da alteração da condição do bloco *if*

Na na terceira linha da Tabela 7, dentre as ações reportadas para o *patch*, a mais importante deixou de ser contabilizada pelo *ADD*: *Loop Condition Change*. Conforme apresentado na Figura 9, a diferença entre as linhas 7 e 8 do *snippet* está em dois lugares: instância da variável *i* que foi modificada e isso foi reportado corretamente pelo *ADD*. Já a alteração realizada na condição do *loop* que passou de $\boldsymbol{i < weights.length}$ para $\boldsymbol{i < begin + length}$ não foi reportado pelo *ADD* e deveria ter sido reportada como *Loop Condition Change*.

```
6           double sumWts = 0;
7   -       for (int i = 0; i < weights.length; i++) {
8   +       for (int i = begin; i < begin + length; i++) {
9               sumWts += weights[i];
10          }
```

Figura 9 – Math_41: Detalhes da alteração na condição do *loop for*

Dos itens analisados manualmente em *Defects4J*, alguns o *ADD* apresentou ações ou padrões de reparo que não existem no *patch*. As Tabelas 8 e 9 demonstram quais fo-

ram esses itens. Por exemplo, no *bug* Time_6, possui um *patch* relativamente grande em termos de métricas, envolvendo: adição de 26 novas linhas de código, modificação de 2 classes e 3 métodos, nenhuma linha existente foi alterada e nenhuma linha foi removida. Entretanto o *ADD* apresenta em seu resultado que uma estrutura condicional foi removida: *Condition Branch Remove*, o que é incoerente em relação às próprias métricas das mudanças realizadas no *patch*.

Tabela 8 – Ações de reparo que segundo a análise manual não existem

Ação de Reparo	**Projeto**	**Identificador**
Method Call Move	Chart	Chart_15
Method Call Parameter Value Change	Math	Math_34
Condition Branch Add	Time	Time_6
Var Replace Var	Time	Time_22
Var Type Change	Math	Math_31
Method Call Parameter Add	Time	Time_22

Tabela 9 – Padrões de Reparo que segundo a análise manual não existem

Padrão de Reparo	**Projeto**	**Identificador**
Constant Change	Chart	Chart_17
Wraps If Else	Math	Math_60

As Tabelas 10 e 11 apresentam o top 5 de ações e padrões de reparo em *Defects4J*. A soma das quantidades de ações de reparo extraídas pelo *ADD* é 3.959, enquanto a soma das quantidades de cada um dos padrões detectados é 1.482 para os 395 *patches* analisados pelo *ADD*, o que permite concluir que em média há aproximadamente 10 ações de reparo e 4 padrões de reparo por *patch*.

Tabela 10 – Ações de reparo que são recorrentes em Defects4J

Ação de Reparo	**Ocorrências**
Method Call Add	807
Assign Add	394
Method Call Parameter Value Change	319
Conditional Branch If Add	315
Variable Add	247

O Top 5 de ações em *Defects4J* somam 2.082, o que é equivalente a 52% do total reportado pelo *ADD*. Se fosse considerado o Top 6 para esse *benchmark*, o sexto mais recorrente seria *Method Call Removed*, que ocorre 239 vezes. Com isso a soma restante de todas as outras 44 ações é igual a 1.843. Logo conclui-se então que as mudanças de métodos são relevantes e caracterizam o *benchmark Defects4J*.

Tabela 11 – Padrões de reparo que são recorrentes em Defects4J

Padrão de Reparo	**Ocorrências**
Copy/Paste	196
Conditional Block Others Add	182
Conditional Block Return Add	137
Expression Logic Modification	112
Single Line	96

Para os padrões de reparo em *Defects4J*, o Top 5 representa 48% da soma total de todos os padrões identificados pelo *ADD* neste *benchmark*. É marcante em *Defects4J* a ocorrência de padrões relacionados a adição de estruturas condicionais, ficando assim caracterizado este *benchmark* em termos de padrões de reparo.

5.3 Bugs.jar

Neste *benchmark* foram analisados manualmente 21 *patches*. A Tabela 5 mostra as ações que o *ADD* deixou de reportar em alguns *patches*.

Tabela 12 – Ações de reparo que o ADD não identificou em Bugs.jar

Ação de Reparo	**Projeto**	**Identificador**
Loop Condition Change	Maven	8cb04253
Return Expression Modification	Jackrabbit-oak	0c3e3d70
Assign Add	Maven	0f3d4d24

O terceiro caso apresentado na Tabela 5 apresenta uma característica semelhante à reportada na Seção 5.1 sobre *Java Annotations*, como apresentado na Figura 10, linha 9.

Além disso, as alterações no *patch* ocorreram também diretamente na alteração da expressão de retorno que começa na linha 10 e termina linha 12, modificação que não foi reportada pelo *ADD*. Para este caso o *ADD* deveria ter sinalizado a ação de reparo *Return Expression Change* que ocorreu uma vez.

```
7   -    boolean includes(Revision r) {
8   -        return high.compareRevisionTime(r) >= 0
9   +    boolean includes(@Nonnull Revision r) {
10  +        return high.getClusterId() == r.getClusterId()
11  +                && high.compareRevisionTime(r) >= 0
12  |       |       && low.compareRevisionTime(r) <= 0;
```

Figura 10 – *Patch* de *Bugs.jar* com modificações em anotações

As Tabelas 13 e 14 apresentam o Top 5 para as ações e padrões de reparo que são mais recorrentes em *Bugs.jar*. Os 1.158 de *Bugs.jar* totalizaram a soma total de 21.652

ações divido entre as 50 ações reportadas por *ADD* e um total de 6.078 padrões através da soma dos 26 diferentes tipos previstos.

Conforme apresentado nas Seções 5.1 e 5.2, o Top 5 de ações e padrões representa em torno de 50% da soma de todas as demais ações e padrões. Em *Bugs.jar* não é diferente, logo pode-se caracterizar que o *dataset* também é representado por alterações relacionadas a chamada de métodos, em termos de ações.

Tabela 13 – Ações de reparo que são recorrentes em Bugs.jar

Ação de Reparo	**Ocorrências**
Method Call Add	4.304
Method Call Parameter Value Change	2.441
Assign Add	1.777
Method Call Remove	1.331
Variable Add	1.210

Semelhante ao que ocorre em *Defects4J*, *Bugs.jar* pode ser caracterizado pela ocorrência de alterações em blocos condicionais. Dos 5 padrões mais recorrentes, adicionar um novo bloco e remover um bloco condicional somam juntos um valor maior que o primeiro item que é o *Copy Paste*. Vale ressaltar que se o Top 6 fosse considerado, o sexto item também estaria relacionado a blocos condicionais.

Tabela 14 – Padrões de reparo que são recorrentes em Bugs.jar

Padrão de Reparo	**Ocorrências**
Copy Paste	1.049
Conditional Block Others Add	774
Wrong Var Reference	489
Expression Logic Modification	387
Conditional Block Remove	367

5.4 Considerações Finais

Diante dos resultados apresentados no Capítulo 5, obtém-se as respostas para as duas *Perguntas de Pesquisa* elaboradas no Capítulo 4 e sugestões de melhorias que podem ser implementadas em trabalhos futuros.

- **PP#1: Qual é a acurácia do *ADD*?**

  A análise manual mostrou que em cada um dos três *benchmarks* houveram um número significativamente baixo de *patches* que o *ADD* deixou de considerar ações e padrões de reparo. Por exemplo, em *Bears* foram 4 *patches*, o que representa 20%

da amostra analisada do *dataset*, porém isso não representa que o *ADD* falhará em 20% dos casos.

Ao todo foram analisados manualmente 79 diferentes *patches*, onde o *ADD* reportou 931 ações de reparo e 330 padrões de reparo. Pelo menos 12 ações de reparos deixaram de ser reportadas pelo *ADD*, o que representa cerca de 1% das ações de reparo dos *patches* da amostra, o que permite concluir que nos testes realizados o *ADD* acertou 99% dos casos.

Em relação aos padrões de reparo, *ADD* falhou relativamente menos do que as ações, logo é afirmativo que menos de 1% dos padrões deixou de ser reconhecido pelo *ADD*, visto que cada um dos *benchmarks* pouco apresentou falhas para este grupo de características.

A Tabela 15 mostra que existem ações de reparo que são mais susceptíveis a não serem encontradas pelo *ADD* do que outras.

A coluna *AM* indica a quantidade de ocorrências que a **A**nálise **M**anual detectou que a ação não foi encontrada, enquanto a coluna *AD* indica quantas vezes a ação foi reportada pelo **AD**D nos mesmos *patches* que foram analisados manualmente e a coluna *TF(%)* indica a taxa em que a ação de reparo deixou de ser identificada.

Tabela 15 – Ações de Reparo que não foram identificadas pela ferramenta *ADD*

Ação de Reparo	**AM**	**AD**	**TF(%)**
Loop Condition Change	3	2	**60.0**
Return Expression Change	3	11	**21.4**
Condition Expression Modification	1	5	**16.6**
Assign Expression Change	1	7	**12.5**
Var Replace Var	1	31	**3.1**
Method Call Remove	1	33	**2.9**
Assign Add	1	75	**1.3**
Method Call Parameter Value Change	1	118	**0.8**

É evidente que a ação *Loop Condition Change* está em uma condição de não ser identificada superior as demais ações implicando que a ferramenta *ADD* é mais propensa a não informar ao pesquisador que houveram mudanças em condições de *loops*. No geral, a Tabela 15 mostra que ações que ocorrem menos são mais propensas a falha.

Conclusão: a ferramenta*ADD* está mais propensa a errar ações que são menos frequentes e mais apto é identificar ações mais frequentes.

- **PP#2: Quais são os padrões e ações de reparos que caracterizam cada um dos *benchmarks*?**

Para caracterizar os *benchmarks* em relação às ações e padrões de reparo que o *ADD* detectou ao longo dos experimentos, foram utilizadas as Tabelas 5, 6, 10, 11, 13 e 14. Os resultados apresentados nos próximos parágrafos não consideram apenas o item que mais foi reconhecido em cada *dataset*, mas sim o agrupamento de ações e padrões que possuem a mesma natureza. Por exemplo: adicionar uma chamada de método ou remover uma chamada de método significa modificar chamadas de métodos, logo o resultado pode ser generalizado em mudança de chamadas de métodos.

Todos os três *benchmarks* apresentaram características semelhantes em relação as ações e padrões de reparo mais reconhecidos pelo *ADD*.

Em termos de ações, as que mais prevaleceram foram as **modificação de chamadas de métodos**, isso inclui qualquer contagem dos 8 tipos diferentes de ações "***Method Call***", porém as ações que tiveram valores significativos para este resultado são: ***Method Call Add, Method Call Remove e Method Call Parameter Value Change***.

Em termos de padrões de reparo os *benchmarks* também apresentaram valores significativos relacionadas a **adição de blocos condicionais**, com destaque para os padrões: ***Conditional Block Others Add, Conditional Block Return Add e Conditional Block Return Add***.

Alguns detalhes que cabem melhorias foram implementados ao longo do trabalho e podem ser implementadas no futuro em *ADD*: reconhecer anotações (Figuras 7 e 10) e identificar encapsulamentos, como o que ocorre na Figura 11.

```
6     public Iterator<Chromosome> iterator() {
7   -     return chromosomes.iterator();
8   +     return getChromosomes().iterator();
9     }
```

Figura 11 – *Patch* com modificações em variáveis que foram encapsuladas

Por fim, é cabível uma correção na identificação dos padrões que o ADD não identificou ao longo dos experimentos, em destaque para as ações de reparo que aparecem em pelos menos dois dos *benchmarks*: *Loop Condition Modification* e *Return Expression Modification* para que o *ADD* não os deixe de reportar.

6 Conclusão

Neste trabalho foi investigado o problema da caracterização de *benchmarks* de *bugs* sob a perspectiva do auxílio de uma ferramenta automática com o enfoque em caracterização e acurácia dos resultados apresentados pela ferramenta.

Madeiral et al. (2018) propuseram *ADD*, a ferramenta utilizada nesta investigação. Até então, apenas *Defects4J* foi caracterizado no mesmo estudo em que *ADD* foi apresentado. *ADD* foi re-documentado com o intuito de demonstrar as principais características no código-fonte que permitem pesquisadores incrementar a ferramenta com novas ações e padrões de reparo.

Os três *benchmarks* foram caracterizados com o auxílio de *ADD Bears*, *Defects4J* e *Bugs.jar*. Estes foram escolhidos pois são trabalhos recentes e reproduzíveis com projetos populares de código aberto. Os sistema de software documentados são escritos na linguagem Java, que é a única linguagem suportada por *ADD* (JUST; JALALI; ERNST, 2014), (MADEIRAL et al., 2019), (SAHA et al., 2018).

Os resultados mostraram que todos os *benchmarks* apresentaram características semelhantes em termos de padrões e ações de reparo. Além disso, observou-se que *ADD* pode ser utilizada na tarefa de caracterizar diferentes benchmarks com uma confiabilidade aceitável.

Como trabalhos futuros sugere-se a caracterização de outros *benchmarks* o que ajuda a melhorar o desempenho do *ADD* e catalogar esses *benchmarks* para outros estudos; inclusão de novas funcionalidades em *ADD* como a que o faça reconhecer anotações (*Java Annotations*) em termos de adição, exclusão e atualização; e por fim, um trabalho contínuo para melhorar o desempenho da ferramenta, principalmente em relação às características que ele venha a demonstrar menos precisão, como as apresentadas neste trabalho.

Referências

DURIEUX, T. et al. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 302–313. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3338911>. Citado na página 11.

FALLERI, J. et al. Fine-grained and accurate source code differencing. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. [s.n.], 2014. p. 313–324. Disponível em: <http://doi.acm.org/10.1145/2642937.2642982>. Citado na página 20.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2014. (ISSTA 2014), p. 437–440. ISBN 978-1-4503-2645-2. Disponível em: <http://doi.acm.org/10.1145/2610384.2628055>. Citado 6 vezes nas páginas 9, 14, 24, 25, 29 e 36.

MADEIRAL, F. et al. Towards an automated approach for bug fix pattern detection. *VI Workshop on Software Visualization, Evolution and Maintenance (VEM)*, 2018. Disponível em: <https://arxiv.org/pdf/1807.11286.pdf>. Citado 5 vezes nas páginas 9, 16, 17, 25 e 36.

MADEIRAL, F. et al. Bears: An extensible java bug benchmark for automatic program repair studies. In: IEEE. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2019. p. 468–478. Citado 6 vezes nas páginas 9, 15, 24, 26, 28 e 36.

MONPERRUS, M. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 234–242. ISBN 978-1-4503-2756-5. Disponível em: <http://doi.acm.org/10.1145/2568225.2568324>. Citado na página 9.

NIDHRA, S.; DONDETI, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, v. 2, n. 2, p. 29–50, 2012. Citado na página 27.

PAWLAK, R. et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, Wiley-Blackwell, v. 46, p. 1155–1179, 2015. Disponível em: <https://hal.archives-ouvertes.fr/hal-01078532/document>. Citado na página 20.

SAHA, R. K. et al. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. New

York, NY, USA: ACM, 2018. (MSR '18), p. 10–13. ISBN 978-1-4503-5716-6. Disponível em: <http://doi.acm.org/10.1145/3196398.3196473>. Citado 5 vezes nas páginas 9, 14, 24, 25 e 36.

SOBREIRA, V. et al. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In: *Proceedings of SANER*. [S.l.: s.n.], 2018. Citado 5 vezes nas páginas 9, 10, 16, 17 e 27.

WEIMER, W. et al. Automatic program repair with evolutionary computation. *Commun. ACM*, ACM, New York, NY, USA, v. 53, n. 5, p. 109–116, maio 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1735223.1735249>. Citado na página 13.

XUAN, J. et al. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, v. 43, n. 1, p. 34–55, Jan 2017. ISSN 0098-5589. Citado 2 vezes nas páginas 13 e 14.

ZHI, J. et al. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, v. 99, p. 175 – 198, 2015. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121214002131>. Citado na página 19.