Sílvio José de Souza

Avaliação e Integração de Ferramentas para Detecção de Defeitos

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Dezembro 2009

Resumo

Recentemente, tem crescido o interesse tanto acadêmico como industrial pelo emprego de técnicas e ferramentas para detecção de defeitos. Em vez de procurarem verificar se um sistema atende a sua especificação, ferramentas para detecção de defeitos – também chamadas de *bug findings tools* – funcionam procurando por violações de padrões de programação. Como exemplo de defeitos detectados por essas ferramentas, podemos citar acesso a referências *null*, uso inapropriado de métodos (como *equals*, *clone* etc), uso incorreto de primitivas de sincronização, *overflow* em vetores, divisão por zero etc. Em resumo, tais ferramentas ampliam e sofisticam a capacidade de detecção de defeitos normalmente realizada por compiladores tradicionais. Esta dissertação tem como objetivo avaliar e propor uma solução para integrar ferramentas para detecção de defeitos. Na primeira parte da avaliação realizada – que teve por objetivo verificar se ferramentas para detecção de defeitos são efetivas em descobrir defeitos reportados por usuários finais – utilizou-se um conjunto de *bugs* catalogados em um repositório denominado iBugs. Esse repositório tem como objetivo exatamente disponibilizar *benchmarks* para avaliação de ferramentas de detecção de defeitos. Na segunda parte da avaliação, realizada com o objetivo de verificar a efetividade de ferramentas para detecção de defeitos na localização de defeitos removidos em versões futuras de um sistema, executou-se a ferramenta FindBugs sobre um determinado conjunto de versões publicadas dos sistemas Rhino e Ajc. Por último, descreve-se na dissertação o projeto e a implementação da meta-ferramenta Smart Bug Detector. A ideia de se construir essa ferramenta surgiu da observação de que ferramentas para detecção de defeitos possuem em comum um pequeno conjunto de detectores, ou seja, o uso combinado dessas ferramentas pode aumentar a taxa de *bugs* reais detectados. Outra motivação para o desenvolvimento do Smart Bug Detector foi a constatação de que utilizar mais de uma ferramenta para detecção de defeitos de forma manual implica em lidar com problemas como aumento de custo de configuração e execução, duplicidade de defeitos detectados e diferentes formatos de relatórios de *bugs*.

Palavras-chave: Qualidade de software. Análise estática de código. Ferramentas para detecção de defeitos.

Abstract

Recently, we have observed a growing interest in techniques and tools to detect defects, both by the academia and by the industry. Rather than trying to determine whether a system meets its specification, tools to detect defects – also called bug findings tools – work by looking for violations of coding standards. As an example of possible defects detected by these tools we can mention access to null references, inappropriate use of methods (such as equals, clone etc), incorrect use of synchronization primitives, overflow in mathematical operations, division by zero etc. In summary, bug finding tools contribute to extend and improve the warning messages typically generated by compilers. They can also be used to enforce coding style guidelines, such as indentation and naming conventions. In this master dissertation we have evaluated and proposed a solution to integrate bug finding tools. In the first part of the evaluation described in the dissertation – which aims to assess whether bug finding tools are effective to find defects reported by end-users – we have relied on a set of bugs catalogued in a repository called iBugs. This repository aims to provide benchmarks for the assessment of tools designed to find defects. In the second part of the assessment, we have evaluated the effectiveness of bug finding tools in detecting defects that have been removed in future versions of the target systems. In the presented evaluation, we have used the FindBugs tool as a representative of current tools to detect defects. We have executed FindBugs over two medium-sized systems: ajc (an AspectJ compiler) and Rhino (a Javascript interpreter). The second part of the dissertation describes the design and implementation of a meta bug finding tool, called Smart Bug Detector. The idea to build this tool came from the observation that tools to detect defects have in common a small set of bug detectors. Therefore, the combined use of these tools can increase the actual rate of bugs found.

Keywords: Software quality. Static analysis of code. Bug finding tools.

Dedico esta dissertação à memória de meu pai, que tudo fez pelos filhos,
e à minha mãe que até hoje nos apóia em tudo.

Agradecimentos

Agradeço, a Deus, que me permitiu chegar até aqui.
Agradeço a minha família pelo apoio e paciência.
Agradeço ao professor Marco Túlio, meu orientador, pela dedicação, paciência e excelente trabalho realizado.
Agradeço ao João Eduardo Mantandon a ajuda indispensável durante a realização dos experimentos.
Agradeço a Giovana Silva, pela atenção e prestatividade.
Agradeço a todos os professores do curso pelos conhecimentos transmitidos.

Conteúdo

Lista de Figuras **vi**

Lista de Tabelas **viii**

1 Introdução **1**

1.1 Visão Geral do Problema 1

1.2 Objetivos 3

1.3 Visão Geral da Solução Proposta 3

1.4 Estrutura da Dissertação 5

2 Revisão da Literatura **7**

2.1 Ferramentas para Deteção de Defeitos 7

2.1.1 FindBugs 9

2.2 Avaliação Empírica de Ferramentas para Detecção de Defeitos 12

2.2.1 Trabalho de Wagner et al. 13

2.2.2 Trabalho de Nathaniel Ayewah et al. 15

2.2.3 Trabalho de Wagner et al. 16

2.2.4 Trabalho de Zheng et al. 18

2.2.5 Trabalho de Spacco et al. 22

2.2.6 Trabalho de Kim et al. 23

2.3 Linguagens para especificação de ***bug patterns*** 24

2.4 Considerações Finais 29

3 Avaliação de Ferramentas para Detecção de Defeitos **31**

3.1 Introdução 31

3.2 Configuração do Experimento 32

3.3 Questão Q1: Ferramentas para Detecção de Defeitos Encontram Defeitos que Darão Origem a Falhas? . . . 33
3.3.1 Resultados . . . 38
3.4 Questão Q2: Ferramentas para Deteção de Defeitos Encontram Defeitos Removidos em Versões Futuras? . . . 41
3.4.1 Defeitos Avaliados . . . 42
3.4.2 Resultados . . . 44
3.5 Riscos à Validade do Estudo de Caso . . . 46
3.6 Conclusões . . . 48

4 A Ferramenta Smart Bug Detector 50
4.1 Visão Geral . . . 50
4.2 Funcionalidades Básicas . . . 52
4.2.1 Pré-requisitos . . . 52
4.2.2 Configuração de Ferramentas para Detecção de Defeitos . . . 53
4.2.3 Cadastro de Prioridades e Classes de Defeitos . . . 54
4.2.4 Composição de *Bug Patterns* . . . 56
4.2.5 Criação e Execução de um Projeto para Detecção de Defeitos . . . 58
4.2.6 Métricas . . . 59
4.3 Arquitetura Interna . . . 60
4.4 Estudo de Caso . . . 66
4.5 Considerações Finais . . . 69

5 Conclusões 71
5.1 Visão Geral da Solução Proposta . . . 71
5.2 Comparação com Trabalhos Relacionados . . . 72
5.3 Contribuições . . . 74
5.4 Trabalhos Futuros . . . 74

Bibliografia 76

Lista de Figuras

1.1 Exemplo de *bug* detectado pelo FindBugs no Eclipse (versão 3.0) 2
1.2 Exemplo de *bug* detectado pelo PMD no Eclipse (versão 3.0) 2

2.1 Exemplo de código com *bug* da categoria *Malicious code vulnerability* . . . 9
2.2 Interface gráfica do FindBugs 12
2.3 Exemplo de expressão que procura três `if` consecutivos 26
2.4 Exemplo de expressão que procura erro de uso do operador '=' 26
2.5 Exemplo de expressão que procura chamada de funções 26
2.6 Exemplo de expressão que procura por sequências de comandos `if` 26
2.7 Exemplo de uso de curingas nomeados 27
2.8 Exemplo de regra do JavaCOP 28
2.9 Exemplo de predicado do JavaCOP 28
2.10 Visão geral da implementação do JavaCOP 29

3.1 Exemplo de comandos para extrair e compilar versões do iBugs 33
3.2 Descrição do *bug* 36234 do Ajc 34
3.3 Código com a versão antes da correção do *bug* 36234 do Ajc 35
3.4 Código com versão da correção do *bug* 36234 do Ajc 36
3.5 Descrição do *bug* 179068 do sistema Rhino 36
3.6 Descrição do *bug* 107299 do Ajc 37
3.7 Descrição do *bug* 193555 do sistema Rhino 38
3.8 Classe antes da correção da falha 193555 38
3.9 Classe após a correção da falha 193555 39
3.10 Defeitos associados a falhas do interpretador Rhino 40
3.11 Defeitos associados a falhas do compilador Ajc 40

3.12 Exemplo de *Bug* detectado na versão 1.51 do Rhino 45
3.13 Exemplo de correção de *bug* realizado na versão 1.52 do Rhino 46

4.1 Visão geral da interface proposta 53
4.2 Exemplo de configuração de conexão da aplicação com banco de dados . . 54
4.3 Configuração das ferramentas para detecção de defeitos 54
4.4 Cadastro de prioridades de *bug patterns* 55
4.5 Cadastro de classes de defeitos 55
4.6 Classificação de defeitos 56
4.7 Composição `Alias` 56
4.8 Composição `Or` 57
4.9 Composição `And` 57
4.10 Composição de *bug patterns* 59
4.11 Tela para criar ou alterar um projeto de detecção de defeitos 59
4.12 Arquitetura da ferramenta Smart Bug Detector 60
4.13 Diagrama de classes da camada de apresentação 61
4.14 Diagrama de classes da camada de domínio 62
4.15 Diagrama de classes da camada de repositório 62
4.16 Diagrama de classes da camada de modelo 63
4.17 Diagrama ER da solução 65
4.18 Diagrama de sequência do fluxo de detecção de defeitos 66

Lista de Tabelas

2.1 Categorias de defeitos do FindBugs 10
2.2 Defeitos removidos no projeto X 14
2.3 Defeitos removidos no projeto Y 15
2.4 Classificação de defeitos detectos no Sun JDK 16
2.5 Sistemas analisados 17
2.6 Eficiência de remoção de defeitos por categoria 18
2.7 Dados do experimento 20
2.8 Taxa de remoção de defeitos por diferentes técnicas 21
2.9 Curingas utilizados para representação de expressões 25
2.10 Curingas nomeados 27
2.11 Exemplos de consultas JTL 27

3.1 Classificação das falhas reportadas 34
3.2 Detecção de defeitos em classes modificadas para correção de falhas 39
3.3 Versões do Rhino analisadas 41
3.4 Versões do Ajc analisadas 42
3.5 Total de *bugs* reportados pelo FindBugs no Rhino (coluna A), total de *bugs* em elementos renomeados no programa (coluna B) e total de *bugs* considerados no estudo (coluna C) 43
3.6 Total de *bugs* reportados pelo FindBugs no Ajc (coluna A), total de *bugs* em pacotes externos (coluna B), total de *bugs* em elementos renomeados no programa (coluna C) e número total de *bugs* considerados no estudo (coluna D) 43
3.7 Tempo de vida de defeitos no interpretador Rhino 44
3.8 Tempo de vida de defeitos no compilador Ajc 45

4.1 *Bugs* detectados com o uso do Smart Bug Detector no sistema rhino 1.43 . 68
4.2 Exemplo de composições 69
4.3 *Bugs* detectados após o cadastro de composições 69

Capítulo 1

Introdução

1.1 Visão Geral do Problema

Recentemente, tem crescido o interesse tanto acadêmico como industrial pelo emprego de técnicas e ferramentas para detecção de defeitos [Lou06, FHP07, AHM+08]. Em vez de procurarem verificar se um sistema atende a sua especificação, ferramentas para detecção de defeitos – também chamadas de *bug findings tools* – funcionam procurando por violações de padrões de programação. Como exemplo de defeitos detectados por essas ferramentas, podemos citar acesso a referências *null*, uso inapropriado de métodos (como *equals*, *clone* etc), uso incorreto de primitivas de sincronização, *overflow* em vetores, divisão por zero etc. Em resumo, tais ferramentas ampliam e sofisticam a capacidade de detecção de defeitos normalmente realizada por técnicas como inspeção e testes.

Dentre as diversas ferramentas para detecção de defeitos existentes, podem ser citados os sistemas Lint [Joh77] e PREfix/PREfast [LBD+04] (para programas em C/C++); FindBugs [HP04], PMD [Cop05], QJ–Pro [QP09] e KlocWork [KNB08] (para programas em Java) e FxCop [Mic09] (para programas em .NET). O processo de detecção de defeitos realizado por essas ferramentas pode ser realizado sobre o código fonte ou sobre o *bytecode* do sistema alvo. Nesse processo, a detecção de *bugs* é realizada por um conjunto de detectores implementados pelas ferramentas. Desta forma, cada detector funciona como um filtro especializado em identificar pontos de código que representam uma violação ou um defeito.

A Figura 1.1 apresenta um exemplo de *bug* identificado pelo detector NP_ALWAYS_NULL do FindBugs no sistema Eclipse (versão 3.0). Esse detector procura por pontos de código com acesso a referências *null*. No exemplo apresentado, observa-se que na linha 131 a chamada a `c.isDisposed()` somente ocorrerá quando a primeira expressão for verdadeira, causando um erro do tipo `NullPointerException`. De fato, verificou-se que

em uma versão posterior, os desenvolvedores do Eclipse corrigiram esse *bug* trocando o operador `&&` pelo operador `||`.

```
127: private void setSmartButtonVisible(boolean visible) {
.....
131: if (c == null && c.isDisposed())
132: return;
.....
142: }
```

Figura 1.1: Exemplo de *bug* detectado pelo FindBugs no Eclipse (versão 3.0)

A Figura 1.2 apresenta um exemplo de *bug* detectado pelo PMD, também na versão 3.0 do Eclipse. Nesse exemplo, o detector do PMD de nome `EmptyMethodInAbstractClass_ShouldBeAbstract` gera um alerta indicando que um método vazio (linhas 422-423) em uma classe abstrata deve ser reescrito como um método abstrato. De fato, verificou-se que a alteração recomendada foi realizada na versão 3.4.

```
62: abstract class AbstractInfoView extends ...{
...
422: protected void internalDispose() {
423: }
...
496: }
```

Figura 1.2: Exemplo de *bug* detectado pelo PMD no Eclipse (versão 3.0)

Apesar do crescente interesse por técnicas automatizadas de detecção de defeitos, duas questões ainda permanecem em aberto:

- Não existe um consenso sobre a real utilidade de tais ferramentas em atividades de manutenção de software.
- Não se sabe ao certo o quanto essas ferramentas podem contribuir para a qualidade final de um software se comparadas a técnicas tradicionais de detecção de defeitos, tais como inspeção e testes.

Observou-se ainda que apesar das ferramentas para detecção de defeitos existentes possuírem um mesmo princípio de funcionamento, elas possuem um pequeno conjunto de detectores em comum. Por exemplo, Rutar et al. encontraram uma correlação de 31% entre os detectores do FindBugs e PMD [RA04]. Desta forma, o uso combinado de duas ou mais ferramentas para detecção de defeitos pode aumentar a taxa defeitos reais localizados [WAWS08, RA04].

Por outro lado, o uso combinado dessas ferramentas só é viável com uma solução que permita realizar a integração e ao mesmo tempo proporcione funcionalidades que resolvam ou minimizem os problemas gerados pela combinação de múltiplas ferramentas para detecção de defeitos, como por exemplo, duplicidade de defeitos detectados e diferentes formatos de relatório de *bugs*.

1.2 Objetivos

Esta dissertação possui dois objetivos principais:

- Avaliar ferramentas para detecção de defeitos: Ferramentas para detecção de defeitos reportam um número elevado de alertas. Assim, nesta dissertação, pretende-se avaliar a efetividade dessas ferramentas, verificando-se a relevância dos defeitos reportados. A ferramenta FindBugs será utilizada nos experimentos realizados de forma a obter dados para responder às seguintes questões motivadoras:

  1. Ferramentas para detecção de defeitos são efetivas para descobrir defeitos que darão origem a falhas reportadas por usuários finais de sistemas de software?
  2. Ferramentas para detecção de defeitos são capazes de indicar defeitos que posteriormente serão removidos manualmente em versões futuras de um sistema de software?

- Projetar e implementar uma solução para integração de ferramentas para detecção de defeitos: O uso combinado dessas ferramentas pode aumentar o potencial de remoção de defeitos. Por outro lado, problemas como alta taxa de falsos positivos, relatórios de *bugs* reportados em formatos distintos e customização complexa podem desestimular a adoção dessas ferramentas. Desta forma, descreve-se nessa dissertação o projeto e a implementação de uma meta-ferramenta para detecção de defeitos, chamada Smart Bug Detector. Essa ferramenta permite o uso integrado de duas ou mais ferramentas para detecção de defeitos, fornecendo funcionalidades que solucionam ou minimizam os problemas observados.

1.3 Visão Geral da Solução Proposta

Avaliação de Ferramentas para Detecção de Defeitos: Objetivando responder às duas questões descritas no primeiro item da Seção 1.2, desenvolveu-se um estudo de caso utilizando a ferramenta FindBugs, uma das mais usadas para deteção de defeitos em

programas Java. A ferramenta FindBugs foi utilizada para detectar defeitos em dois sistemas: Rhino (um interpretador de JavaScript) e Ajc (o compilador de AspectJ mais usado atualmente).

Na primeira parte do estudo – que teve por objetivo verificar se ferramentas para detecção de defeitos são efetivas em descobrir defeitos reportados por usuários finais – utilizou-se um conjunto de *bugs* catalogados em um repositório denominado iBugs [DZ07]. Esse repositório tem como objetivo exatamente disponibilizar *benchmarks* para avaliação de ferramentas de detecção de defeitos. No iBugs, estão armazenadas as versões antes e após a correção de cada *bug* reportado por usuários finais, bem como uma descrição do *bug* e exemplos para reprodução do mesmo. Nesta dissertação, as descrições dos *bugs* disponibilizados no repositório foram lidas e analisadas objetivando descartar as falhas consideradas *solicitações de funcionalidades não implementadas ou implementadas de forma parcial*. Sobre o conjunto de falhas restante, classificadas como *defeitos*, as seguintes ações foram tomadas:

- As versões antes e após a correção foram extraídas do repositório.
- Com o auxílio de uma ferramenta para cálculo de diferença textual, identificou-se quais classes foram alteradas para a correção do *bug*.
- Executou-se a ferramenta FindBugs sobre a versão antes da correção da falha a fim de verificar se os *bugs* reportados por essa ferramenta estavam relacionados com os pontos alterados para a correção do *bug* reportado pelo usuário.

Desta forma, nessa etapa do estudo, foram obtidos dados que permitiram reponder à primeira questão proposta.

Na segunda parte do estudo, desenvolvida com o objetivo de verificar a efetividade de ferramentas para detecção de defeitos em indicar defeitos corrigidos em versões futuras, executou-se a ferramenta FindBugs sobre um determinado conjunto de versões publicadas dos sistemas Rhino e Ajc. Nessa análise, considerou-se somente os *bugs* de alta prioridade reportados pelo FindBugs.

Para o conjunto de versões utilizadas no estudo, coletou-se os *bugs* de alta prioridade reportados pelo FindBugs e procurou-se, a cada versão analisada, determinar quais *bugs* eram novos, quais eram ocorrências de versões anteriores e quais foram corrigidos. Assim, com esse procedimento foram obtidos dados que permitiram tirar conclusões sobre a segunda questão proposta.

Integração de Ferramentas para Detecção de Defeitos: Na segunda parte da dissertação, descreve-se o projeto e a implementação da meta-ferramenta Smart Bug Detector criada com o objetivo de permitir a integração de ferramentas para detecção de defeitos. Basicamente essa ferramenta se propõe a resolver ou minimizar os seguintes problemas:

- Dificuldade para uso de duas ou mais ferramentas para detecção de defeitos: O uso combinado de duas ou mais ferramentas para detecção de defeitos pode contribuir para aumentar a taxa de verdadeiros positivos detectados [WAWS08, RA04].

- Grande número de defeitos reportados: Ferramentas de deteção de defeitos geram um grande número de defeitos normalmente reportados em formato XML ou diretamente na interface gráfica da ferramenta. Com a solução proposta pretende-se consolidar os defeitos reportados em um único formato, bem como facilitar o compartilhamento dos resultados produzidos.

- Eficiência do critério de priorização de defeitos: O critério de priorização de defeitos identificados por ferramentas para detecção de defeitos nem sempre é eficiente ou adequado a um determinado sistema [KE07]. A ferramenta Smart Bug Detector permite redefinir a prioridade e a descrição de um *bug* ou combinar dois ou mais *bugs* em composições do tipo `And` e `Or`.

Finalmente, realizou-se um estudo de caso com a ferramenta Smart Bug Detector configurada para executar as ferramentas FindBugs e PMD. O estudo de caso teve por objetivo demonstrar os benefícios proporcionados pela ferramenta.

1.4 Estrutura da Dissertação

O restante desta dissertação está organizado da seguinte maneira:

- No Capítulo 2, apresenta-se uma introdução sobre detecção de defeitos por meio de ferramentas, os conceitos básicos da área e as principais ferramentas existentes. Também apresenta-se de forma mais aprofundada a ferramenta FindBugs, uma das mais utilizadas em Java. Por fim, este capítulo apresenta alguns dos principais trabalhos que tratam do emprego de ferramentas para detecção de defeitos e trabalhos que abordam linguagens que podem ser utilizadas para detecção de defeitos.

- No Capítulo 3, relata-se um estudo de caso elaborado com os seguintes objetivos:

  - Avaliar o grau de efetividade de ferramentas para detecção de defeitos reportados por usuários finais.

  - Avaliar se defeitos de alta prioridade detectados pelo FindBugs são de fato removidos em versões futuras de um sistema.

- No Capítulo 4, apresenta-se o projeto e a implementação de uma meta-ferramenta desenvolvida com o objetivo de integrar ferramentas para detecção de defeitos.

- Por fim, no Capítulo 5, são apresentadas as considerações finais do estudo realizado e da ferramenta desenvolvida, as contribuições da dissertação e propostas de trabalhos futuros.

Capítulo 2

Revisão da Literatura

Este capítulo está organizado como descrito a seguir. A Seção 2.1 apresenta uma introdução sobre ferramentas para deteção de defeitos, os conceitos básicos da área e as principais ferramentas existentes. A Seção 2.1.1 introduz de forma mais aprofundada a ferramenta FindBugs, uma das mais utilizadas em Java. A Seção 2.2 apresenta alguns dos principais trabalhos que tratam do emprego de ferramentas para detecção de defeitos. A Seção 2.3 apresenta alguns dos principais trabalhos sobre linguagens que podem ser utilizadas para especificação de *bug patterns*. Finalmente, na Seção 2.4, são apresentadas algumas considerações finais sobre as ferramentas e os trabalhos discutidos neste capítulo.

2.1 Ferramentas para Deteção de Defeitos

Recentemente, tem crescido o interesse tanto acadêmico como industrial pelo emprego de técnicas e ferramentas para detecção de defeitos [Lou06, FHP07, AHM+08]. Em vez de procurarem verificar se um sistema atende a sua especificação, ferramentas para detecção de defeitos - também chamadas de *bug findings tools* - funcionam procurando por violações de padrões de programação. Como exemplo de defeitos detectados por essas ferramentas, podemos citar acesso a referências *null*, uso inapropriado de métodos (como *equals*, *clone* etc), uso incorreto de primitivas de sincronização, *overflow* em vetores, divisão por zero etc. Em resumo, tais ferramentas ampliam e sofisticam as mensagens de *warning* normalmente emitidas por compiladores.

Adicionalmente, podem contribuir para verificar estilos e boas práticas de programação, como convenções de nome e de indentação. Dentre as diversas ferramentas para detecção de defeitos existentes, podem ser citados os sistemas Lint [Joh77] e PREfix/PREfast [LBD+04] (para programas em C/C++); FindBugs [HP04], PMD [Cop05] QJ Pro e KlocWork (para programas em Java) e FxCop (para programas em .NET).

A seguir são apresentadas as principais características de algumas das ferramentas mencionadas anteriormente:

1. FindBugs – É uma ferramenta *open source* desenvolvida por pesquisadores da universidade de Maryland. Realiza deteção de defeitos em programas Java por meio de inspeção de *bytecodes*. Ela é baseada no conceito de *bug patterns* e possui detectores que procuram por padrões que com frequência indicam um erro. A arquitetura de *plugins* do FindBugs permite a adição de novos detectores criados em Java [HP04]. A ferramenta pode ser executada por meio de interface gráfica, interface texto ou ainda com *plugin* para o ambiente Eclipse.

2. PMD – A ferramenta PMD é voltada para detecção de defeitos através da inspeção do código fonte. A ferramenta contém detectores para verificação de estilo de código (criação de variáveis desnecessárias, blocos *try-catch* vazios, etc) e defeitos (conexão aberta, *null pointer*, etc). A ferramenta pode ser executada como um *plugin* para diversos editores como Eclipse, JBuilder e JDeveloper dentre outros ou também por meio de linha de comando. A ferramenta PMD permite a adição de novas regras por meio de programação em Java ou criando uma expressão *XPath*.

3. FxCop – É uma ferramenta para detecção de defeitos voltada para análise de código compilado em .NET. Possui um conjunto de detectores que procuram por padrões considerados defeitos ou práticas não–recomendadas de programação nessa plataforma. O FxCop pode ser executado por meio de uma interface gráfica ou linha de comando. Assim como o FindBugs e PMD, o FxCop também permite a adição de novos detectores. A ferramenta pode ser obtida e utilizada livremente, no entanto o código fonte não é fornecido.

4. KlocWork – É uma ferramenta comercial para detecção de defeitos em sistemas escritos em C, C++, C# e Java.

5. QJ Pro – É uma ferramenta *open source* para detecção de defeitos em sistemas Java com características semelhantes à ferramenta PMD. Possui uma interface gráfica e *plugins* para os editores JBuilder, JDeveloper e Eclipse. Assim como no PMD, essa ferramenta também inspeciona o código fonte.

2.1.1 FindBugs

FindBugs é uma das ferramentas para detecção de defeitos mais utilizadas na plataforma Java, possuindo usuários como Google, Sun, dentre outros. O princípio de funcionamento da ferramenta é baseado no conceito de *bug patterns*, isto é, um padrão de código que na maioria das situações representa um erro. O FindBugs inspeciona o *bytecode* Java para detecção de *bug patterns*, o que significa que o código fonte não é necessário. No entanto, a presença do fonte permite a rastreabilidade entre os defeitos reportados e o trecho de ocorrência dos mesmos. A ferramenta foi desenvolvida por Bill Pugh e David Hovemeyer. Atualmente ela é mantida por Bill Pugh e voluntários [HP04].

A ferramenta utiliza as seguintes categorias para classificação de defeitos:

1. *Bad practice* – Violação de práticas de código recomendadas. Exemplos deste tipo de violação incluem métodos que ignoram retorno de funções e nomes de classes, métodos e atributos fora do padrão recomendado, etc.

2. *Correctness* – Erro de código com grande probabilidade de ser um defeito. Os detectores desta categoria trabalham procurando por padrões que normalmente indicam um erro de programação. Alguns exemplos são: *loops* infinitos, passagem de parâmetros com valor *null* e métodos com retorno sempre igual a falso. Esta categoria tem baixa taxa de falsos positivos.

3. *Malicious code vulnerability* – *Bug patterns* nesta categoria indicam defeitos que podem comprometer a execução ou expor informações internas da aplicação quando a mesma for manipulada por código malicioso. Como exemplo de código com *bug* desta categoria, no método `consulta` da Figura 2.1, o FindBugs irá apontar um defeito informando que o método estático expõe uma representação interna ao retornar um *array*.

```
public class Mapper {
  private static String[] _nomes;
  public static String[] consulta() {
     return _nomes;
  }
}
```

Figura 2.1: Exemplo de código com *bug* da categoria *Malicious code vulnerability*

4. *Multithreaded correctness* – Os *bug patterns* nesta categoria indicam defeitos em classes e métodos relacionados à execução de múltiplas *threads*.

5. Desempenho – Indicam código que pode comprometer o desempenho da aplicação. Como exemplo de defeitos nesta categoria cita-se concatenação de *strings* no interior de *loops* e alocação desnecessária de tipos primitivos.

6. *Security* – Defeitos relacionados a segurança, como por exemplo, senha de banco de dados em branco ou como constante dentro da aplicação e uso de expressões SQL dinâmicas.

7. *Dodgy* – Código confuso, anômalo ou escrito de forma a conduzir a erros. Como exemplo de defeitos nesta categoria cita-se blocos de comandos vazios, comparação desnecessária com *null* envolvendo variáveis que não aceitam esse valor e métodos muito extensos que prejudicam o entendimento e manutenção.

8. Internacionalização – Esta categoria agrupa detectores que procuram por violações de práticas recomendadas para desenvolvimento de software com suporte a múltiplos idiomas. Até a versão 1.3.8 existia apenas um *bug pattern* nesta categoria.

9. Experimental – Esta categoria foi criada para agrupar detectores em fase de avaliação.

Em levantamento realizado na documentação da versão 1.3.8, verificou-se um total de 369 *bug patterns* que a ferramenta é capaz de detectar. A classificação desses *bug patterns* nas categorias descritas anteriormente é mostrada na Tabela 2.1.

Categoria	**Total**
Bad practice	81
Correctness	131
Experimental	1
Internationalization	1
Malicious code vulnerability	12
Multithreaded correctness	40
Performance	26
Security	9
Dodgy	58
Total	**369**

Tabela 2.1: Categorias de defeitos do FindBugs

A ferramenta oferece suporte a filtros e classificação de defeitos. Equipes de qualidade podem utilizar esses recursos para analisar os defeitos reportados em uma versão e classificá-los, de forma a repassar para a equipe de desenvolvimento somente os itens relevantes.

Quando uma nova versão for avaliada, a aplicação de filtros permite a visualização somente dos itens novos. Assim, a atenção da equipe de qualidade e desenvolvimento pode ser focada somente nestas ocorrências. Os defeitos analisados podem ser classificadas em: *Needs further study*, *Not a bug*, *Most harmeless*, *Should fix*, *Must fix*, *Bad analyses* e *Unclassified*.

O FindBugs suporta duas técnicas de rastreamento de defeitos entre versões [SHP06]. A primeira técnica é denomida *Paring*. Esta técnica agrupa os defeitos de ambas as versões e utiliza a assinatura dos defeitos para determinar se dois defeitos de versões distintas são idênticos. A segunda técnica denominada *Warning Signatures* monta um *string* contendo o nome da classe, do campo e o método de ocorrência do defeito. Esse *string* é então utilizado para gerar um código *Hash MD5*. Desta forma, se defeitos de diferentes versões possuírem o mesmo código *hash* eles são considerados idênticos.

Outro recurso do FindBugs é a capacidade de ajuste do nível de precisão da análise. O nível de precisão utilizado está diretamente relacionado à taxa de falsos positivos gerados. Os níveis suportados são:

1. *Low* – Reporta todos os tipos de defeitos. Consequentemente, o número de falsos positivos é maior.

2. *Medium* – Reporta somente defeitos de média e alta prioridade. Normalmente, as ferramentas para detecção de defeitos associam uma prioridade mais alta aos defeitos que consideram mais importantes para correção [KE07]. As prioridades são determinadas por heurísticas existentes em cada detector [APM+07].

3. *High* – Reporta somente defeitos de alta prioridade. Este nível possui menor número de falsos positivos.

4. *Relaxed* – Neste modo a ferramenta desabilita heurísticas usadas para evitar falsos positivos.

Execução da Ferramenta – A ferramenta possui duas interfaces, uma interface gráfica e uma interface texto [HP04]. Em ambiente Windows a execução pode ser realizada invocando o *script* de execução `findbugs.bat` com os parâmetros desejados para indicar modo de execução (gráfico ou texto), forma de apresentação dos resultados, nível de precisão da ferramenta, dentre outros. Por fim, a execução do FindBugs em modo texto permite que o resultado da análise possa ser gravado em arquivos XML ou HTML.

A Figura 2.2 mostra a interface gráfica da ferramenta organizada em três paineis. O primeiro painel é divido em duas partes. A da esquerda apresenta os defeitos detectados

organizados em uma árvore organizada por critérios como prioridade, classe e categoria. A segunda parte do primeiro painel mostra o trecho de código onde o defeito foi detectado (supondo que o código fonte esteja disponível). O painel central apresenta as características do defeito, como a classe, método e linha de ocorrência. Essa informação é sempre exibida mesmo sem a presença do código fonte. O terceiro painel mostra a descrição do defeito e um texto orientando como resolver o tipo de problema reportado.

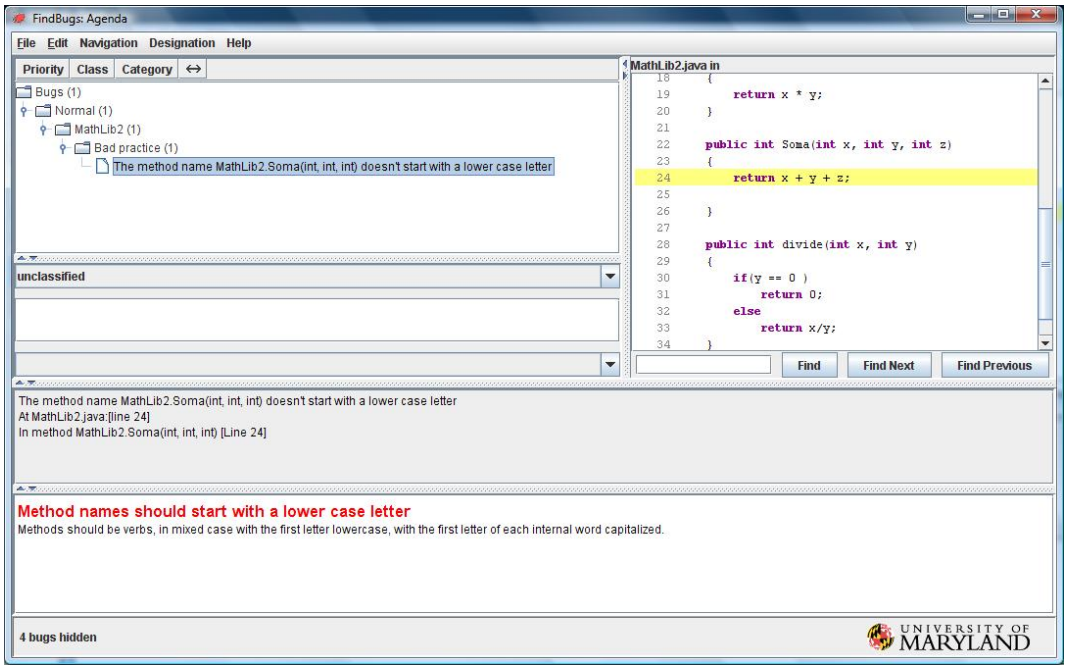

Figura 2.2: Interface gráfica do FindBugs

2.2 Avaliação Empírica de Ferramentas para Detecção de Defeitos

Nesta seção serão apresentados alguns trabalhos que avaliam ferramentas para detecção de defeitos, comparam a sua efetividade com outras técnicas e permitem tirar conclusões sobre o uso dessas ferramentas no processo de desenvolvimento de software.

2.2.1 Trabalho de Wagner et al.

Introdução: Neste trabalho, Wagner, Aichner, Wimmer e Schawalb realizam estudos em dois softwares industriais que por questões de confidencialidade foram denominados de projeto X e projeto Y [WAWS08] . Sobre estes dois projetos, duas ferramentas baseadas em *bug patterns* para Java foram avaliadas: FindBugs e PMD. O objetivo principal do trabalho foi investigar como ferramentas para detecção de defeitos podem contribuir para garantia da qualidade no processo de desenvolvimento de software. Para alcançar o objetivo, três questões foram formuladas na pesquisa:

1. Quantos defeitos de campo a ferramenta precisa detectar para ser economicamente viável?[1]

2. Ferramentas baseadas em *bug patterns* podem detectar defeitos de campo ou defeitos detectados por outras técnicas tais como inspeção e testes?

3. Classes mais propensas a defeitos podem ser identificadas pelas ferramentas?

Metodologia utilizada: O trabalho empregou a seguinte metodologia:

1. Foram utilizados dois sistemas de suporte a vendas desenvolvidos pela empresa alemã Softlab. O primeiro sistema, projeto X, possui 600 KLOC e 2900 classes. O segundo sistema, projeto Y, possui 40 KLOC e 250 classes.

2. As ferramentas FindBugs e PMD foram aplicadas em ambos os sistemas com dois conjuntos de configurações: padrão e customizada. O modo padrão compreende as configurações pré-definidas pelo fabricante da ferramenta. O modo customizado compreende configurações realizadas no estudo de forma a reduzir o número de falsos positivos.

Projeto X - Avaliação de diferentes versões: Após avaliar cinco diferentes versões do projeto X desenvolvidas ao longo de 2,5 anos foram identificados 67 defeitos removidos de uma versão para outra. Nesta avaliação, as ferramentas foram executadas com as configurações customizadas. O estudo procurou determinar as causas da remoção dos defeitos. As seguintes categorias foram utilizadas para classificar os motivos da remoção:

1. Falha – O defeito foi removido porque realmente conduziu a uma falha visível para o usuário do sistema.

[1]Neste trabalho, os autores usam o termo *defeito de campo* como sinônimo de falha.

2. Mudança – O defeito foi removido devido a uma mudança no código não relacionada com um defeito em si. Exemplos são mudanças em classes e migração de versões de Java.

3. Erro da ferramenta – Defeitos que não foram reportados novamente pela ferramenta embora ainda presentes nas classes. A causa normalmente está em alterações no código que tornaram o defeito "invisível"para a ferramenta.

4. Desconhecido – Defeitos onde a causa da remoção não foi identificada.

Utilizando estes critérios os defeitos removidos foram categorizados conforme apresentado na Tabela 2.2. Os resultados mostram que nenhum defeito foi removido devido a uma falha percebida pelo usuário.

Versões		**Falha**	**Mudança**	**Erro Ferramenta**	**Desconhecido**	**Total**
a	b	0	11	9	2	22
b	c	0	22	3	2	27
c	d	0	8	4	0	12
d	e	0	5	1	0	6
Total		0	46	17	4	67

Tabela 2.2: Defeitos removidos no projeto X

Projeto X - Avaliação de defeitos de campo: O objetivo desta avaliação é determinar quantos defeitos de campo podem ser localizados por ferramentas para detecção de defeitos. Foram avaliados defeitos reportados ao longo de 4 anos. De um total de 615 defeitos reportados, 99 foram selecionados aleatoriamente. Dentre esses 99 defeitos, 27 defeitos foram descartados por não terem sua resolução relacionada a um ponto de código específico. Restaram desta forma 72 defeitos para avaliação. Sabendo o ponto de código onde o defeito foi localizado, executou-se as ferramentas para detecção de defeitos na versão correspondente a fim de verificar se tal defeito poderia ser detectado automaticamente.

Verificou–se que nenhum defeito reportado pelo FindBugs ou PMD está relacionado com os 72 defeitos de campo selecionados. Um dos motivos apresentados foi que, dentre os defeitos de campo avaliados, 47 defeitos foram classificados como sendo sendo defeitos lógicos tais como uso incorreto de APIs, uso incorreto de constantes e tamanho de fonte. Claramente, estas categorias de defeitos não podem ser detectadas por ferramentas baseadas em *bug patterns*.

Projeto Y - Avaliação de diferentes versões: Neste projeto, foram avaliados seis versões desenvolvidas ao longo de seis meses. A Tabela 2.3 apresenta os resultados obtidos. Pode-se observar que de um total de 24 defeitos removidos de uma versão para outra, apenas quatro foram relacionados a defeitos percebidos pelo usuário. Este projeto, por não estar em produção no momento da pesquisa não teve uma segunda avaliação sobre defeitos de campo.

Versões		**Falha**	**Alteração**	**Erro Ferramenta**	**Total**
a	b	1	2	5	8
b	c	3	10	0	13
c	d	0	0	0	0
d	e	0	3	0	3
e	f	0	0	0	0
Total		4	15	5	24

Tabela 2.3: Defeitos removidos no projeto Y

Conclusões: Segundo os autores, os resultados obtidos nos projetos X e Y sugerem que ferramentas para detecção de defeitos baseadas em *bug patterns* não são efetivas para detectar defeitos de campo. O principal motivo é que estes defeitos são em sua maioria de natureza lógica, ou seja, relacionados às regras de negócio do sistema.

2.2.2 Trabalho de Nathaniel Ayewah et al.

De acordo com Ayewah, Pugh, Morgenthaler, Penix e Zhou, existe pouca informação na literatura sobre a precisão dos defeitos reportados por ferramentas para detecção de defeitos [APM+07]. Neste trabalho, eles avaliaram a ferramenta FindBugs realizando experimentos com três sistemas: Sun JDK 1.6.0, Sun GlassFish J2EE Server e parte do Google CodeBase. Os objetivos principais do trabalho foram:

1. Avaliar os tipos de defeitos reportados pelo FindBugs e sua classificação em falsos positivos, triviais e sérios.

2. Verificar as razões pelas quais ferramentas para detecção de defeitos frequentemente reportam defeitos triviais.

Resultados obtidos na avaliação do projeto Sun JDK: Foram analisados 89 *releases* públicas (*builds*) da plataforma JDK considerando somente defeitos de alta e média prioridade da categoria corretude. A Tabela 2.4 apresenta os resultados obtidos classificados por relevância. Pode-se observar que dentre os 379 defeitos detectados, 38 (10%)

se mostraram realmente sérias.

Classificação	**JDK**
Análise incorreta	5
Sem impacto	160
Algum Impacto	176
Sério	38
Total	379

Tabela 2.4: Classificação de defeitos detectos no Sun JDK

Resultados obtidos na avaliação do projeto GlassFish: A execução do FindBugs sobre este sistema também considerou somente defeitos da categoria corretude de alta e média prioridade. Para este caso, o estudo procurou identificar as remoções de defeitos entre versões. Foram observados 58 defeitos removidos, sendo que 50 foram devido a defeitos reportados pelo FindBugs e outras oito devido a edições ocorridas por outros motivos.

Resultado obtidos na avaliação do Google CodeBase: Os autores também relatam a experiência de execução do FindBugs no Google CodeBase analisando defeitos de corretude de média/alta prioridade. Os defeitos foram classificados como falso positivo, trivial ou defeito. Foi verificado tempo médio de cinco meses para correção de um defeito reportado.

Conclusões: Os autores concluem que um dos motivos pelos quais ferramentas para detecção de defeitos reportam defeitos reais, porém triviais, é que tais ferramentas não conhecem o que o código deve fazer. Desta forma, elas não são capazes de verificar se o código está corretamente criado para fazer o que se espera. Eles concluem ainda que a existência de defeitos de baixo impacto não é uma barreira para adoção de ferramentas para detecção de defeitos e que uma substancial proporção dos defeitos encontrados por essas ferramentas tem impacto funcional.

2.2.3 Trabalho de Wagner et al.

Neste trabalho, Wagner, Jurjens, Koller e Trischberger afirmam que a automatização obtida por ferramentas para detecção de defeitos pode reduzir o tempo gasto em testes e inspeção de código [WJK+05]. A dificuldade consiste em entender como os defeitos

localizados por ferramentas para deteção de defeitos estão relacionado com defeitos localizados por outras técnicas. Este estudo aborda essa dificuldade oferecendo respostas para três questões:

1. Quais tipos de defeitos são detectados por diferentes técnicas (ferramentas para detecção de defeitos, inpeção e testes)?

2. Existem sobreposição de defeitos detectados?

3. Qual a taxa de falsos positivos das ferramentas para detecção de defeitos?

Metodologia: Neste estudo, os autores utilizaram cinco projetos, sendo quatro deles de uma empresa de telecomunicações e um projeto acadêmico em fase final de testes. Todos os projetos foram implementados em Java e suas características de número de classes e tamanho estão sumarizadas na Tabela 2.5. O estudo utilizou três ferramentas para detecção de defeitos: FindBugs, PMD e QJ Pro.

Projeto	**Classes**	**Tamanho**	**Tipo**
A	1066	58	Industrial
B	215	24	Industrial
C	–	3	Industrial
D	572	34	Industrial
EstA	28	4	Acadêmico

Tabela 2.5: Sistemas analisados

Em todos os projetos foram aplicadas técnicas de ferramentas para detecção de defeitos e teste. Somente no projeto C foi possível realizar inspeção. No estudo, os autores utilizaram cinco categorizações de defeitos (onde 1 é o nível mais severo):

1. Defeitos que causam queda na aplicação.

2. Defeitos que causam uma falha lógica na aplicação.

3. Defeitos com informação insuficiente de erro.

4. Defeitos que violam boas práticas com impactos em desempenho.

5. Defeitos que reduzem a legibilidade e manutenibilidade do código.

Esta classificação foi adotada para ajudar a relacionar os defeitos detectados pelas diferentes técnicas utilizadas no estudo.

Resultados obtidos: Os defeitos detectados por cada técnica foram coletados e categorizados, conforme mostrado na Tabela 2.6. O valor percentual indica a eficiência de cada técnica por categoria. Com base nesses resultados, pode-se observar que para os defeitos mais severos, categorias 1 e 2, as técnicas de testes e inspeção se mostraram mais eficiente, enquanto que para defeitos de nível crítico mais baixo, a técnica que usa ferramentas para detecção de defeitos apresentou melhores resultados.

Categoria	**Análise estática**	**Inspeção**	**Testes**	**Total**
1	22% (8)	35% (13)	43% (16)	100% (37)
2	15% (4)	50% (13)	35% (9)	100% (26)
3	85% (40)	0% (0)	15% (7)	100% (47)
4	70% (32)	30% (14)	0% (0)	100% (46)
5	82% (501)	18% (112)	0% (0)	100% (613)
Total	585	152	31	769

Tabela 2.6: Eficiência de remoção de defeitos por categoria

Conclusões: Os resultados obtidos neste estudo comparativo mostram que:

1. Testes e inspeção não podem ser substituídos por ferramentas para detecção de defeitos devido às diferenças de tipos de defeitos que cada técnica é capaz de detectar. Os resultados obtidos no trabalho demonstram que a maioria dos defeitos detectados por testes estão relacionados a problemas nas regras de negócio da aplicação (defeitos de lógica e tratamento de mensagens insuficiente). Os defeitos localizados por inspeção e ferramentas para detecção de defeitos estão em sua maioria relacionados a erros de programação como, por exemplo, conexão com banco de dados não fechada, variáveis não inicializadas e *null pointers*.

2. Análise estática pode ser um bom estágio pré-inspeção, pois consegue evitar a detecção manual de certos tipos de defeitos.

2.2.4 Trabalho de Zheng et al.

Neste trabalho Zheng, Williams, Nagappan, Hudepohl e Vouk procuram determinar como ferramentas para detecção de defeitos podem ajudar organizacões a melhorar de forma econômica a qualidade de seus produtos de software [ZWN+06]. O estudo foi realizado avaliando defeitos de três grandes produtos de software escritos em C/C++, sobre os quais foram aplicadas detecção de defeitos e inspeção. O objetivo da pesquisa foi dividido em sete questões:

Q1: O emprego de ferramentas para detecção de defeitos é uma forma econômica de localização de defeitos?
Q2: O produto terá qualidade superior se ferramentas para detecção de defeitos fizer parte do processo de desenvolvimento?
Q3: Qual o grau de efetividade de ferramentas de detecção de defeitos em localizar defeitos comparado a inspeção e teste?
Q4: Ferramentas para detecção de defeitos podem ser úteis para identificar módulos com problemas?
Q5: Quais classes de defeitos são mais frequentemente localizados por ferramentas para detecção de defeitos, por inspeção e por teste? Quais classes de defeitos não são reportados pelos clientes, ou seja, não são percebidos pelo cliente durante o uso do sistema?
Q6: Quais tipos de erros de programação são mais frequentemente localizados por ferramentas para detecção de defeitos?
Q7: Ferramentas para detecção de defeitos podem ser utilizadas para localizar erros de programação que tenham potencial para causar vulnerabilidades de segurança?

Metodologia: No estudo, dados de três grandes sistemas desenvolvidos pela Nortel Networks foram coletados e analisados. Os dados consistiram de defeitos reportados por equipes de inspeção e teste compostas por 200 pessoas. Também foram considerados os defeitos reportados pelos clientes dos sistemas. O total de linhas de código foi de aproximadamente 3000 KLOC. A classificação de defeitos utilizada foi a *Orthogonal Defect Classification* da IBM, cujo objetivo é categorizar defeitos de modo que cada tipo de defeito seja associado a uma fase do processo de desenvolvimento [CBC+92].

A maioria das análises apresentadas foram baseadas nos dados obtidos pela ferramenta FlexeLint, apesar de outras como Klockwork e Reasoning Illuma serem também usadas pela Nortel. O motivo é que a quantidade de defeitos reportados pela ferramenta FlexeLint foi cerca de duas vezes maior que a quantidade reportada pelo Klockwork e quatro vezes maior que o reportado pelo Reasoning.

Conforme mostrado na Tabela 2.7, os sistemas foram nomeadas de A a C. Sobre os sistemas A e B somente a técnica de detecção de defeitos foi utilizada. Três *releases* do sistema C foram analisados, sendo que na *release* 0 somente foi realizada inspeção e nos *releases* 1 e 2 foram realizados inspeção e deteção de defeitos.

Resultados: Os resultados obtidos foram analisados de acordo com as questões propostas:

Produto/Release	Análise Estática	Inspeção
A	FlexeLint, Klockwork	–
B	FlexeLint	–
C0	–	Sim
C1	FlexeLint, Reasoning, Klockwork	Sim
C2	FlexeLint, Klockwork	Sim

Tabela 2.7: Dados do experimento

Resultado de Q1: O emprego de ferramentas para detecção de defeitos é uma forma econômica de localização de defeitos? Para responder a essa questão foi computado primeiramente o custo de remoção de defeitos por inspeção. O cálculo desse custo foi realizado somando o tempo gasto nas atividades multiplicado pelo salário dos envolvidos e dividido pelo número de defeitos detectados. Em seguida, foi calculado o custo da técnica de detecção de defeitos. Esse custo foi calculado somando-se o custo de licença da ferramenta mais o custo de análise para se eliminar falsos positivos divido pelo número de defeitos verdadeiros restantes. O custo estimado para a técnica de detecção de defeitos foi da mesma ordem da técnica de inspeção, concluindo-se que detecção de defeitos possui um custo acessível.

Resultado de Q2: O produto terá qualidade superior se ferramentas para detecção de defeitos forem incorporadas ao processo de desenvolvimento? A qualidade final dos produtos avaliados foi medida com base no número de defeitos reportados por teste e por clientes. No entanto, a análise realizada para verificação dessa questão não foi conclusiva devido à grande variação de valores entre os produtos.

Resultado de Q3: Qual o grau de efetividade de ferramentas para detecção de defeitos em localizar defeitos comparado a técnicas de inspeção e teste? As métricas utilizadas nesta análise foram quantidade de defeitos detectados por ferramentas para detecção de defeitos, por inspeção e por testes. A eficiência de cada técnica para remoção de defeitos foi medida dividindo o número de defeitos detectados por determinada técnica pelo total geral de defeitos considerando o uso de todas as técnicas. A Tabela 2.8 apresenta os resultados obtidos. Verificou-se que ferramentas para detecção de defeitos e inspeção estão na mesma ordem de eficiência. No entanto, a remoção de defeitos com base em testes é da ordem de duas a três vezes maior que a técnica de ferramentas para detecção de defeitos.

Resultado de Q4: Ferramentas para detecção de defeitos podem ser úteis para identificar módulos com problemas? Para essa análise foi utilizado o produto B, por ser o

Sistema	Análise Estática(%)	Inspeção(%)	Teste(%)
A	23,39	Não realizado	Não realizado
B	Não realizado	Não realizado	Não realizado
C0	Não realizado	39,55	96,73
C1	31	20,48	98,18
C2	36,53	33,21	62,57

Tabela 2.8: Taxa de remoção de defeitos por diferentes técnicas

único com uma clara divisão de módulos. Os defeitos reportados por testes e pelos clientes do produto foram classificadas por módulo, assim como os defeitos detectados por ferramentas para detecção de defeitos. Os valores obtidos indicaram que os módulos mais problemáticos estavam diretamente relacionados com os módulos que apresentaram maior número de defeitos na aplicação de ferramentas para detecção de defeitos. Desta forma, conclui-se que o número de defeitos por módulo detectados por ferramentas para detecção de defeitos pode ser um fator para identificar módulos com problemas em um sistema.

Resultado de Q5: Quais classes de defeitos são mais frequentemente localizados por ferramentas para detecção de defeitos, por inspeção e por teste? Quais classes de defeitos não são reportadas pelos clientes? Nesta análise verificou-se que: a) os tipos de defeitos detectados por ferramentas para detecção de defeitos são em sua maioria *Checking* e *Assignment*, de acordo com a classificação ODC; b) verificou-se que para a técnica de inspeção, os tipos de defeitos predominantes são de Algoritmo, Documentação e Verificação; c) para a técnica de teste, verificou–se que a maioria dos defeitos reportados são do tipo Algoritmo e Função.

Resultado de Q6: Quais tipos de erros de programação são frequentemente localizados por ferramentas para deteção de defeitos? A métrica utilizada nesta análise foi a quantidade de defeitos por categoria. Os resultados indicaram que a grande maioria dos defeitos foi produzida por um pequeno conjunto de tipos de erros de programação. Por exemplo o uso de *null pointer* é o tipo de erro mais frequente, respondendo por cerca de 45% de todos os defeitos. 90% dos defeitos reportados está relacionado a um grupo de dez tipos de erros de programação.

Resultado de Q7: Ferramentas para detecção de defeitos podem ser utilizadas para localizar erros de programação que tenham potencial para causar vulnerabilidades de segurança? A métrica para esta análise foi a quantidade de defeitos detectados por ferramentas para detecção de defeitos classificados por tipo de defeito. Os resultados indicaram

que ferramentas para detecção de defeitos podem ser utilizadas para detecção de defeitos relacionados a vulnerabilidades de segurança.

2.2.5 Trabalho de Spacco et al.

Neste trabalho, Jaime Spacco, David Hovemeyer e Willian Pugh, afirmam que a capacidade de acompanhar a ocorrência de um potencial defeito entre multiplas versões é importante pelos seguintes motivos [SHP06]:

1. Relembrar as condições de código que foi revisado e marcado como correto, apesar da ocorrência de um defeito.

2. Construir um delta de defeitos entre versões mostrando quais defeitos são novos, quais são de versões anteriores e quais foram removidos. Desta forma, equipes de inspeção podem focar somente em novos defeitos.

O principal objetivo deste trabalho foi discutir as duas técnicas existentes no FindBugs para rastreamento de defeitos entre versões, os seus méritos e como elas podem ser utilizadas no processo de desenvolvimento. Também foram apresentados e discutidos os resultados obtidos no rastreamento de defeitos em diferentes versões do Sun JDK. O problema tratado no trabalho pode ser resumido da seguinte forma: após uma versão de software ser analisada, produzindo uma lista de possíveis defeitos, cada um é avaliado e classificado. A classificação pode variar de acordo com a ferramenta utilizada. Quando uma nova versão do sistema for analisada, deseja-se associar os resultados da análise anterior com os da análise atual, de forma que o trabalho de auditoria se concentre somente nos defeitos novos.

As duas técnicas implementadas pela ferramenta FindBugs e analisadas no trabalho foram:

1. *Emparelhamento*: Esta técnica emprega um algoritmo baseado em séries cada vez menos precisas para realizar o casamento de defeitos de duas versões analisadas. Este algoritmo produz um código *hash* para cada defeito e tem como princípio de funcionamento que um defeito do conjunto A é igual a um defeito do conjunto B se seus códigos *hash* forem iguais.

2. Assinaturas de defeitos: Para realizar o casamento de defeitos entre duas versões, esta técnica gera um *string* único obtido a partir do nome da classe, campo e método onde o defeito ocorreu. Um código *hash MD5* é então calculado a partir deste *string* que por sua vez é utilizado como critério de casamento.

Resultados: O trabalho apresenta dados obtidos no rastreamento de defeitos do *Sun JDK* em um conjunto de 116 versões. Observou-se que:

1. A maioria dos defeitos de alta prioridade foram corrigidos.

2. Houve uma tendência de aumento na densidade de defeitos ao longo das versões, que, no entanto, não foi atribuído a uma queda de qualidade e sim a um acúmulo de falsos positivos e defeitos não corrigidos.

3. Especificamente nas classes do pacote `java.util`, cujos desenvolvedores envolvidos utilizam o FindBugs com frequência, observou-se uma densidade de defeitos inferior à média do *Sun JDK*.

Conclusões: Este trabalho mostra que o recurso de correspondência de defeitos entre versões é de fundamental importância para que uma ferramenta para detecção de defeitos seja adotada no ciclo de desenvolvimento de software. As principais aplicações práticas são: supressão de falsos positivo, aplicação de auditoria entre as versões, construção de deltas de advertência e estudo do ciclo de vida de defeitos, permitindo uma visão sobre a evolução do código. O trabalho ainda pondera que com a pouca atenção dada ao recurso de rastreamento de defeitos entre versões (o foco principal está na construção de novos detectores), tem surgido várias abordagens, cada uma com suas próprias vantagens e desvantagens, mas sem nenhum estudo que procure unificá-las em uma interface comum que potencialize as vantagens de cada uma.

2.2.6 Trabalho de Kim et al.

Neste trabalho, Sunghun Kim e Michael D. Ernst, afirmam que ferramentas para detecção de defeitos possuem uma alta taxa de falsos positivos apesar do critério de priorização de defeitos utilizado em que é associada uma prioridade alta para os defeitos considerados mais importantes [KE07]. Eles afirmam ainda que o critério de priorização tende a ser ineficiente com base em observações realizadas em três ferramentas (FindBugs, PMD e JLint), envolvendo testes com três sistemas (Columba, Lucene e Scarab).

Metodologia: Foram realizados dois experimentos usando as ferramentas FindBugs, PMD e JLint sobre os softwares Columba, Lucene e Scarab:

1. No primeiro experimento, mediu-se o percentual de defeitos apontados pelas ferramentas para detecção de defeitos e que foram removidos devido a correções de defeitos no software.

2. No segundo experimento, mediu-se o percentual de verdadeiros positivos dentre os defeitos de alta prioridade.

Eles observaram uma taxa de correção de defeitos de 6%, 9% e 9% respectivamente para os três sistemas analisados em um período de um a quatro anos de desenvolvimento. Ou seja, cerca de 90% dos defeitos reportados pelas ferramentas continuaram no programa ou foram removidos por alterações não relacionadas a correção de defeitos.

Observou-se que a precisão máxima na correção de defeitos reais, com base no critério de alta prioridade, foi de 3%, 12% e 8%. Os valores apresentados dizem respeito somente ao total de defeitos reportados pelas ferramentas para detecção de defeitos, enquanto que no trabalho anterior foram apresentados resultados que comparam a eficiência para remoção de defeitos por meio de ferramentas de detecção em relação às técnicas de inspeção e testes. Esse trabalho também propõe uma técnica de priorização de defeitos que utiliza uma base de dados histórica onde grava-se a experiência de correção de defeitos no software alvo. Um algoritmo realiza busca nesta base de dados para determinar quais defeitos são importantes. O algoritmo baseia-se no critério de que se um defeito de uma determinada categoria foi corrigido, então o tipo de defeito é importante.

Algoritmo: Com base nas observações do primeiro experimento, verificou-se a baixa eficiência no critério de priorização de defeitos utilizado nas ferramentas para detecção de defeitos. O algoritmo proposto realiza extração de dados no histórico de correção de defeitos do software. Uma categoria de defeito recebe um peso maior se a base de dados histórica contém muitas correções para a mesma. Por outro lado, uma categoria recebe um peso baixo se raramente teve um defeito corrigido.

Conclusões: Com o uso do algoritmo de priorização proposto, obteve-se um ganho de 17%, 25% e 67% sobre os softwares *Columba*, *Lucene* e *Scarab*.

2.3 Linguagens para especificação de *bug patterns*

Muitas ferramentas para detecção de defeitos permitem que seus usuários construam e adicionem novos detectores de forma que a ferramenta possa atender a necessidades específicas de um sistema e/ou empresa. Por exemplo, pode-se verificar se as convenções de nome e estilos de programação adotadas na empresa estão sendo seguidas.

No entanto, a construção de novos detectores exige programação complexa e um profundo conhecimento da arquitetura da ferramenta. São discutidos nesta seção alguns trabalhos que propõem linguagens para expressar consultas em código fonte de forma

mais simples, em uma sintaxe próxima da linguagem de programação a ser pesquisada.

Framework para pesquisa de código usando padrões de progamação: Neste trabalho, Paul e Prakah propõem uma linguagem para pesquisa em código fonte usando padrões de programação [PP94]. Um protótipo denominado SCRUPLE foi criado para as linguagens C e PL/AS para validação da abordagem proposta. A ferramenta SCRUPLE funciona da seguinte forma:

1. O código fonte é lido e uma árvore de representação sintática (AST) é gerada.

2. O arquivo contendo os padrões de busca é compilado e para cada padrão de busca é gerado um autômato finito não-determinístico. A entrada para o autômato é a AST gerada no passo 1.

3. Um intepretador simula a execução do autômato na AST produzindo o conjunto de resultados da busca.

A linguagem proposta no trabalho estende a linguagem onde será realizada a busca. Mais especificamente, no trabalho a linguagem *C* foi estendida da seguinte forma:

1. Uso de curingas para representação de expressões. A Tabela 2.9 apresenta os curingas utilizados para representação de entidades no código.

Uso	**Entidades**	**Coleção de entidades**
Declaração	$d	$*d
Tipo	$t	
Variável	$v	$*v
Função	$f	$f
Expressão	#	#*
Comando	@	@*

Tabela 2.9: Curingas utilizados para representação de expressões

   A Figura 2.3 apresenta um exemplo de expressão com curingas. Nesse exemplo, a expressão representa uma busca por ocorrência de três `if` consecutivos. A Figura 2.4, outro exemplo de uso de curinga, representa uma expressão que procura a ocorrência de comandos `if` onde '=' foi usado incorretamente no lugar de '=='.

2. Curingas para representação de padrões de busca por coleções, apresentados na terceira coluna da Tabela 2.9. Como exemplos de aplicação, apresenta-se a expressão da Figura 2.5 que procura todos os comandos que são chamadas de funções e a Figura 2.6, que apresenta uma expressão que procura uma sequência de comandos

```
if # @;
if # @;
if # @;
```

Figura 2.3: Exemplo de expressão que procura três `if` consecutivos

```
if (# = #)@;
```

Figura 2.4: Exemplo de expressão que procura erro de uso do operador '='

com três ou mais comandos `if` entre eles.

```
$f(#*);
```

Figura 2.5: Exemplo de expressão que procura chamada de funções

```
if # @;
@*;
if # @;
@*;
if # @;
```

Figura 2.6: Exemplo de expressão que procura por sequências de comandos `if`

3. Curingas nomeados. A Tabela 2.10 relaciona os curingas nomeados que permitem um maior poder de expressão em relação aos dois tipos anteriores. O uso desses curingas permite por exemplo representar padrões onde um objeto precisa ser referenciado em dois ou mais pontos.

   A Figura 2.7 apresenta um exemplo de expressão que utiliza curingas nomeados. Nesse exemplo, a expressão exibida representa uma consulta que procura por situações onde o valor de duas variáveis são trocados entre si.

4. Recursos avançados. São recursos propostos pela linguagem para representação de padrões avançados de busca como, por exemplo, busca em profundidade em métodos, *loops* e blocos de comando.

Com a linguagem de representação de padrões proposta e a ferramenta de validação SCRUPLE este trabalho demonstrou que fragmentos de código complexos podem ser localizados com relativa facilidade. A linguagen proposta pode ser empregada em diversas

Entidade	Padrão
declaração	$d_{name}
conjunto de declarações	$*d_{name}
tipo	$t_{name}
variável	$v_{name}
conjunto de variáveis	$v*_{name}
função	$f_{name}
expressão	#_{name}
conjunto de expressões	#*_{name}
comando	@_{name}
sequência de comandos	@*_{name}

Tabela 2.10: Curingas nomeados

```
$v_tmp = $v_x;
@*;
$v_x = $v_y;
@*;
$v_y = $v_tmp;
```

Figura 2.7: Exemplo de uso de curingas nomeados

aplicações práticas, como localização de *bug patterns*, uso em atividades de reengenharia de código e compreensão de programas.

***JTL - the Java Tools Language*:** JTL é uma linguagem criada por Cohen, Gil e Maman com o objetivo de permitir busca e seleção de trechos de programas Java [CGM06]. Algumas aplicações práticas citadas são seleção de *join points* para programação orientada a aspectos, como ferramenta de compreensão de programas e como ferramenta de detecção de *bug patterns* ou construções de programação não desejadas.
A sintaxe da linguagem é derivada da linguagem Java. Na Tabela 2.11 são apresentados alguns exemplos de consultas e seu significado.

Consulta	Significado
`public abstract void`	Métodos abstratos, públicos, com retorno *void* e sem parâmetros
`public int`	Campos do tipo *int* de visibilidade pública
`public static main()`	Todos os métodos *main*

Tabela 2.11: Exemplos de consultas JTL

O Interpretador JTL criado para a linguagem permite o uso da ferramenta como um programa *stand-alone* ou como uma API para ser executada a partir de programas Java.

Um *plugin* para o Eclipse foi criado ilustrando a segunda forma de uso citada e permitindo a seleção de trechos de programas através de consultas JTL. Resumindo, a ferramenta JTL se destaca por sua expressividade, facilidade de aprendizado e de integração com outras ferramentas.

Framework para implementação de tipos acopláveis: Neste trabalho Andreae, Noble, Markstrum e Millstein apresentam a solução denominada JavaCOP, um *framework* para desensolvimento de tipos acopláveis para Java [ANMM06]. O objetivo do *framework* é permitir o estabelecimento de restrições para uso de tipos de uma forma declarativa usando uma linguagem de regras. Com a aplicação de restrições, pode-se evitar erros de programação ou não conformidade com padrões de programação estabelecidos.
Uma regra do JavaCop é formada por uma função que inicia com a palavra chave ***rule*** seguida de um nome, um parâmetro e um corpo contendo uma sequência de restrições. A Figura 2.8 exemplifica uma regra que restringe a atribuição de *null* nos objetos onde ela for aplicada.

```
rule checkNonNull2(VarDef v) {
 where(requiresNonNull(v)) {
 require(!v.type.isPrimitive()):
  error(v, "@NonNull can only annotate variables of reference type");
} }
```

Figura 2.8: Exemplo de regra do JavaCOP

A declaração de predicados é um recurso do JavaCop que tem por objetivo auxiliar na construção de regras. A Figura 2.9 apresenta a declaração do predicado *requiresNonNull* utilizado na regra *checkNonNull2*. Este predicado recebe um objeto da AST e verifica se ele possui a anotação *NonNull*.

```
declare requiresNonNull(Tree t) {
  require(t.holdsSymbol && t.getSymbol.hasAnnotation("NonNull"));
}
```

Figura 2.9: Exemplo de predicado do JavaCOP

A Figura 2.10 ilustra o princípio de funcionamento do JavaCOP. A ferramenta incorpora o compilador Java (javac) de forma que produz Java bytecode de programas após checar se não violam as regras estabelecedidas. O primeiro passo realizado quando se

compila um programa com o JavaCop é a compilação das regras, tarefa realizada pelo *JavaCOP.Compiler*. O segundo passo é realizado pelo *JavaCOP.Framework* que aplica as regras compiladas na AST do código fonte do programa para verificar se não há violações. Não tendo sido detectadas violações, o programa é compilado com sucesso.

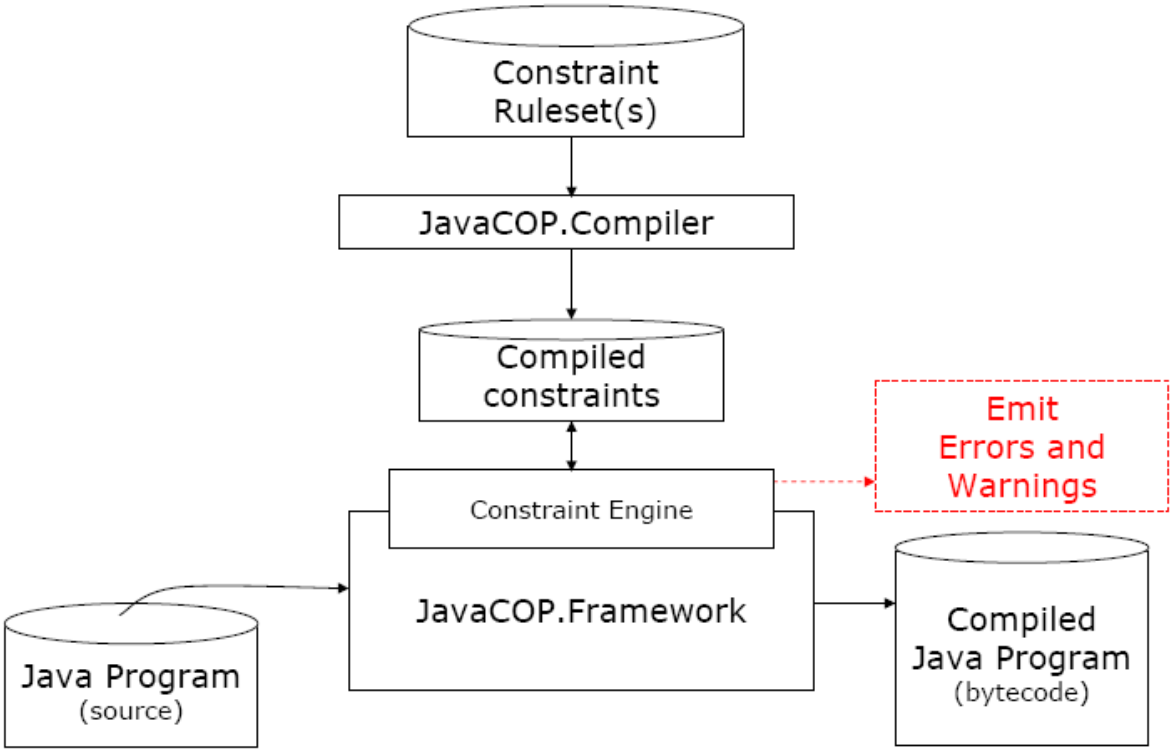

Figura 2.10: Visão geral da implementação do JavaCOP

2.4 Considerações Finais

Os trabalhos apresentados nesta seção fornecem dados importantes para tomada de decisões sobre o uso de ferramentas para detecção de defeitos no processo de desenvolvimento. Verifica-se através destes trabalhos que:

1. Ferramentas para detecção de defeitos podem ser utilizadas durante o ciclo de desenvolvimento. Nesta fase estas ferramentas são efetivas na detecção de erros de programação e violação de boas práticas.

2. Ferramentas para detecção de defeitos devem ser vistas como uma técnica complementar a inspeção e testes. Cada técnica possui um grau de eficiência diferente na

remoção de certos tipos de defeitos.

3. O uso de ferramentas para deteção de defeitos em sistema em produção não é efetivo. Nesta fase, os erros mais comuns estão relacionados às regras implementadas e como as ferramentas para detecção de defeitos não conhecem essas regras, elas não são capazes de detectar esses tipos de erros.

4. O uso eficiente de ferramentas para detecção de defeitos no processo de desenvolvimento deve estar associado a alguma forma de rastreamento de defeitos entre versões. Devido ao grande número de defeitos detectados, é necessário que o trabalho de análise realizado sobre cada defeito seja preservado de uma versão para outra. Desta forma, a equipe envolvida no processo pode focar somente nos novos defeitos apontados a cada versão.

Um ponto negativo de ferramentas para detecção de defeitos é a complexidade exigida para criação de detectores. Os trabalhos aqui apresentados sobre linguagens de especificação de *bug patterns* apresentam formas mais simples para realizar busca em código. No entanto, as ferramentas para detecção de defeitos públicas atualmente disponíveis ainda precisam incorporar de alguma forma os recursos apresentados nesses trabalhos.

Capítulo 3

Avaliação de Ferramentas para Detecção de Defeitos

3.1 Introdução

Apesar do interesse e do número crescente de ferramentas para detecção de defeitos, ainda não existe um consenso sobre a real utilidade de tais ferramentas em atividades de manutenção de software. Assim, neste capítulo relata-se um estudo de caso elaborado com o objetivo de avaliar o grau de efetividade de uma ferramenta para detecção de defeitos. Com essa avaliação, pretende-se verificar a relevância dos defeitos reportados por essas ferramentas.

Na terminologia adotada no estudo, considera-se que um elemento de software não codificado corretamente possui um defeito; quando um resultado incorreto é apresentado para o usuário final em decorrência da execução de um código com defeito considera-se que ocorreu uma falha.

Duas questões centrais questões motivaram o desenvolvimento do estudo de caso:

- (Q1) Ferramentas para detecção de defeitos são efetivas para encontrar defeitos que darão origem a falhas reportadas por usuários finais de sistemas de software?
- (Q2) Ferramentas para detecção de defeitos são capazes de encontrar defeitos que posteriormente serão removidos manualmente em versões futuras de um sistema de software?

A fim de contribuir com respostas para essas perguntas, relata-se neste capítulo uma experiência de aplicação de uma ferramenta para detecção de defeitos em diversas versões de dois sistemas de código aberto. O experimento foi dividido em duas partes com o

objetivo de obter dados que permitam responder às questões Q1 e Q2. A ferramenta escolhida foi o sistema FindBugs [HP04], que é uma das ferramentas mais usadas para detecção de defeitos de programas Java. Neste trabalho, o FindBugs foi usado para detectar defeitos em dois sistemas:

- Rhino[2]: um interpretador de JavaScript com 49 KLOC desenvolvido como parte do projeto Mozilla.
- Ajc[3]: o compilador de AspectJ mais usado atualmente, possuindo na última versão de 2008 cerca de 75 KLOC.

Esses dois sistemas foram escolhidos por dois motivos principais: (1) são sistemas de médio e grande porte, de relativa complexidade; (2) possuem um histórico bem documentado de falhas, publicamente disponível no repositório iBugs [DZ07]. Esse repositório tem como objetivo exatamente disponibilizar *benchmarks* para avaliação de ferramentas de detecção de defeitos.

3.2 Configuração do Experimento

Descreve-se nesta seção a configuração do experimento realizado, incluindo informações sobre a ferramenta FindBugs e sobre os dois sistemas analisados neste trabalho.

Ferramenta para Detecçao de Defeitos: A experiência descrita neste capítulo utilizou a versão 1.3.6 da ferramenta FindBugs. Nessa experiência, o FindBugs foi executado em modo texto com a opção *-high* ligada. Essa opção indica que apenas *bugs* de alta prioridade devem ser apontados, isto é, *bugs* que denotam código com grande probabilidade de possuir defeitos. Além dessa opção, o FindBugs suporta outras categorias de *bugs*, menos severas. No entanto, essas categorias não foram utilizadas, de forma a concentrar em defeitos que possam levar a falhas e, portanto, minimizar o número de falsos positivos apontados pela ferramenta.

Sistemas e Relatórios de Falhas Analisados: O repositório iBugs possui cadastrados 32 *bugs* que usuários reportaram para o interpretador Rhino. Esses *bugs* foram reportados via sistema de rastreamento de *bugs* Bugzilla, normalmente utilizado em sistemas da fundação Mozilla. No iBugs, estão armazenadas as versões antes e após a correção de cada um desses 32 *bugs*.

[2]http://www.mozilla.org/rhino
[3]http://www.eclipse.org/aspectj

Para o compilador Ajc, o repositório iBugs armazena um total de 348 *bugs*, também reportados via Bugzilla por programadores envolvidos no desenvolvimento do compilador. Assim como no Rhino, no repositório iBugs do Ajc são disponilizadas as versões antes e após a correção de cada um desses *bugs*.

Cada versão amazenada no iBugs contém, além das falhas reportadas por usuários, um relatório em formato XML com a identificação e descrição de cada uma das falhas e um arquivo de configuração que permite extrair e compilar versões usando o software `ant`, uma ferramenta de compilação baseada em Java[4].

A manipulação do conteúdo do iBugs é realizada por meio de linhas de comando interpretadas pela ferramenta `ant`. A Figura 3.1 apresenta comandos que exemplificam, respectivamente, como extrair e compilar uma determinada versão contida no repositório iBugs. Nesse exemplo, primeiramente as versões antes e depois da correção da falha de número 157509 são extraídas do repositório (linha 1). Em seguida a versão antes da correção da falha é compilada permitindo o uso do FindBugs para analisar o *bytecode* resultante da compilação (linha 2).

```
1: ant -DfixId=157509 checkoutversion
2: ant -DfixId=157509 -Dtag=pre-fix buildversion
```

Figura 3.1: Exemplo de comandos para extrair e compilar versões do iBugs

As versões armazenadas no iBugs foram utilizadas para responder a questão Q1. Para responder a questão Q2, foram utilizadas as últimas versões estáveis dos dois sistemas, disponibilizadas nos seus *sites* na Web.

3.3 Questão Q1: Ferramentas para Detecção de Defeitos Encontram Defeitos que Darão Origem a Falhas?

Para responder a essa questão, as seguintes atividades foram realizadas:

Filtragem de Falhas: Inicialmente, o texto livre de descrição dos *bugs* reportados via Bugzilla para os sistemas Rhino e Ajc foi lido e analisado. O objetivo foi distinguir entre

[4]http://ant.apache.org/

bugs que representam falhas e *bugs* que na verdade são reclamações sobre funcionalidades ou requisitos implementados de forma parcial. Os casos onde a leitura da descrição do *bug* não permitiu concluir com certeza se o *bug* se tratava de falha ou reclamação sobre funcionalidades não implementadas foram classificados na categoria falha.

Classificação das Falhas: A Tabela 3.1 reporta o número de *bugs* classificados em cada uma das duas categorias mencionadas anteriormente. Evidentemente, apenas *bugs* que representam falhas foram considerados no estudo, visto que não faz sentido esperar que uma ferramenta para detecção de defeitos ajude a descobrir que um determinado requisito ou funcionalidade não foi implementado (ou foi implementado de forma incorreta).

Tipos de Bugs	**Rhino**		**Ajc**	
	Qtd	%	Qtd	%
Implementações parciais	13	40	88	25
Falhas	19	60	260	75
Total	32	100	348	100

Tabela 3.1: Classificação das falhas reportadas

Exemplo de Falha: Como exemplo de *bug* que representa falha, cita-se o *bug* número 36234 reportado para o compilador Ajc. Na Figura 3.2, apresenta-se o texto original do relatório do *bug* obtido do iBugs e na Figura 3.3 apresenta-se o trecho de código antes da correção do *bug*.

```
Getting an out of memory error when compiling with Ajc 1.1 RC1.
I know this is not very descriptive, but maybe you can point me into a
direction of getting more output.

here some additional information though:
Code base is medium size (about 1500 classfiles)
I removed all my aspects and still receive the error.
Running it from the command line:
ajc -classpath whateveritis -sourceroots whateveritis -d whateveritis
```

Figura 3.2: Descrição do *bug* 36234 do Ajc

Na Figura 3.4 apresenta-se a mesma classe alterada para a correção do problema apresentado na Figura 3.3. Por essa figura, conclui-se que para a correção do problema, a classe *Main* foi alterada em dois pontos. Foi adicionado um atributo estático de nome

OUT_OF_MEMORY_MSG (linha 2) contendo uma mensagem que irá avisar ao usuário sobre o problema e orientá–lo sobre como aumentar a memória disponível para o Ajc. Além disso, o método `runMain` foi alterado para tratar a exceção `OutOfMemoryError` de forma a exibir o texto da mensagem sempre que este tipo de exceção ocorrer (linhas 19 a 31).

A extração dos trechos de código modificados para a correção da falha foi realizada com o auxílio da ferramenta WinMerge 2.10[5]. Essa ferramenta compara dois arquivos exibindo a diferença textual entre eles.

```
1: public class Main {
2:    ..
3:    public void runMain(String[] args, boolean useSystemExit) {
4:      ..
5:      run(args, holder);
6:      ..
6:    }
7:  }
```

Figura 3.3: Código com a versão antes da correção do *bug* 36234 do Ajc

Exemplo de Implementação Parcial: Como exemplo de *bug* que representa reclamações sobre funcionalidades ou requisitos implementados de forma parcial, cita-se o *bug* número 179068, reportado para o interpretador Rhino, onde verifica-se que o usuário reclama que o sistema não suporta *strings* de entrada com mais de 64K caracteres. O texto original da descrição do *bug* é apresentado na Figura 3.5. Com o auxílio da ferramenta de WinMerge verificou-se que para a implementação desta funcionalidade, duas classes foram modificadas em pontos diversos, resultando em 130 linhas de código alteradas. Devido ao grande número de linhas alteradas, não apresenta-se nesta seção exemplos de código para essa categoria. No entanto, esse exemplo permite concluir que realmente não se pode esperar que ferramentas para detecção de defeitos sejam capazes de localizar defeitos desta categoria.

Como um segundo exemplo de implementação parcial, dessa vez para o sistema Ajc, cita-se o *bug* de número 107299 apresentado na Figura 3.6. Nesse *bug*, o usuário reclama que o compilador não reconhece caminhos absolutos do Windows nos parâmetros de entrada.

Execução da Ferramenta para Detecção de Defeitos: Para *bugs* classificados como

[5]http://winmerge.org.

```
 1: public class Main {
 2:  private static final String OUT_OF_MEMORY_MSG
 3:   = "AspectJ " + Version.text
 4:   + " ran out of memory during compilation:"
 5:   + LangUtil.EOL + LangUtil.EOL
 6:   + "Please increase the memory available to ajc by
 6:      editing the ajc script " + LangUtil.EOL
 9:   + "found in your AspectJ installation directory.
10:      The -Xmx parameter value" + LangUtil.EOL
12:   + "should be increased from 64M (default) to 128M or even 256M."
14:   + LangUtil.EOL + LangUtil.EOL
15:   + "See the AspectJ FAQ available from the documentation link"
16:   + LangUtil.EOL
17:   + "on the AspectJ home page at http://www.eclipse.org/aspectj";
18:   ..
19:  public void runMain(String[] args, boolean useSystemExit) {
20:    ..
21:  // make sure we handle out of memory gracefully...
22:    try {
23:      run(args, holder);
24:    }
25:    catch (OutOfMemoryError outOfMemory) {
26:      IMessage outOfMemoryMessage = new Message(OUT_OF_MEMORY_MSG,
27:      null,true);
28:     holder.handleMessage(outOfMemoryMessage);
29:     systemExit(holder);
30:     //we can't reasonably continue from this point.
31:    }
32:  }
```

Figura 3.4: Código com versão da correção do *bug* 36234 do Ajc

```
Currently Rhino does not support long string literals with more
then 64K of characters, which can be a problem to run it against
automatically generated scripts.
```

Figura 3.5: Descrição do *bug* 179068 do sistema Rhino

falhas, foram baixadas do repositório iBugs as versões antes e depois da correção da falha. Foram então realizados os seguintes procedimentos:

1. Calculou-se uma diferença textual entre a versão antes e depois da correção da falha, a fim de determinar as classes e métodos da versão antiga que foram editados

```
ajc doesn't recognize Windows absolute file paths that don't
start with a drive letter, e.g., run:
ajc -inpath \test.jar
[error] build config error: bad inpath component: \test.jar

but

ajc -inpath c:\test.jar
works

ajc -aspectpath \test.jar Test.aj

[error] build config error: bad aspectpath: \test.jar

ajc -aspectpath c:\test.jar Test.aj
(works)
```

Figura 3.6: Descrição do *bug* 107299 do Ajc

e/ou modificados para correção da falha. Para cálculo dessa diferença, foi usada a ferramenta WinMerge, citada anteriormente.

2. Executou-se o FindBugs sobre a versão antes da correção da falha. Defeitos apontados em classes modificadas para correção da falha – determinadas no passo anterior – foram coletados; os demais defeitos foram desprezados.

Esse procedimento foi baseado na seguinte hipótese: durante a fase de manutenção de um sistema, uma ferramenta para detecção de defeitos pode ajudar mantenedores a localizar defeitos responsáveis por falhas reportadas por usuários finais quando ela é capaz de apontar *bugs* em classes ou – melhor ainda – em métodos que devem ser modificados para correção dessa falha.

Exemplos: Para exemplificar um caso em que o FindBugs detectou defeitos em métodos efetivamente alterados na versão após a correção da falha, selecionou-se o *bug* 193555 do sistema Rhino. A Figura 3.7 apresenta a descrição desse *bug* no iBugs, onde o usuário relata uma funcionalidade que deixou de funcionar corretamente de uma versão para outra.

A Figura 3.8 apresenta a versão de uma classe antes da correção da falha de número 193555, sendo que a execução do FindBugs detectou nessa classe um *bug* em um método efetivamente alterado para a correção da falha reportada pelo usuário. O *bug* detectado pelo FindBugs nesse caso alerta que a variável local `pIndex` recebe um valor que não é

```
In 1.5R4 a function expression can not access its function name.
To test, run the following test case in the Rhino shell with
arbitrary optimization:

var x = function f() { return f.toString(); };
var ok = (x.toString() === x());
print(ok ? "OK" : "FAILED");

In Rhino 1.5R3 it prints OK while running 1.5R4 produces:
js: "/home/igor/js/x/f_rec.js", line 2: uncaught JavaScript exception:
ReferenceError: "f" is not defined. (/home/igor/js/x/f_rec.js; line 2)
```

Figura 3.7: Descrição do *bug* 193555 do sistema Rhino

utilizado (linha 5). Na versão após a correção da falha, Figura 3.9, observa-se que essa variável foi removida. No entanto, como diversas classes e métodos foram alterados para a correção da falha, não se pode concluir que o uso da ferramenta para detecção de defeitos ajudaria a descobrir o problema.

```
 1:  public class VariableTable {
 2:    public void addParameter(String pName) {
 3:     // Check addParameter is not called after addLocal
 4:      if (varStart != itsVariables.size()) Context.codeBug();
 5:      int pIndex = itsVariableNames.get(pName, -1);
 6:      if (itsVariableNames.has(pName)) {
 7:          String message = Context.getMessage1("msg.dup.parms",
pName);
 8:          Context.reportWarning(message, null, 0, null, 0);
 9:       }
10:      int index = varStart++;
11:      itsVariables.add(pName);
12:      itsVariableNames.put(pName, index);
13:     }
14:  }
```

Figura 3.8: Classe antes da correção da falha 193555

3.3.1 Resultados

A Tabela 3.2 apresenta informações consolidadas sobre os defeitos reportados pelo FindBugs em classes modificadas para correção de falhas. Conforme pode ser observado nesta tabela, dentre as 19 versões analisadas do interpretador Rhino, em apenas 11 versões

```
 1: public class VariableTable {
 2:  public void addParameter(String pName) {
 3:   // Check addParameter is not called after addLocal
 4:   if (varStart != itsVariables.size()) Context.codeBug();
 5:   // Allow non-unique parameter names: use the last occurrence
 6:   int index = varStart++;
 7:   itsVariables.add(pName);
 8:   itsVariableNames.put(pName, index);
 9:  }
10: }
```

Figura 3.9: Classe após a correção da falha 193555

o FindBugs foi capaz de apontar defeitos em classes que foram efetivamente modificadas para correção da falha apontada. Nas outras 8 versões analisadas, nenhum dos erros apontados pelo FindBugs ocorreu em classses modificadas para correção de falhas. No caso do compilador Ajc, os resultados são piores: em apenas 18% das versões analisadas o FindBugs foi capaz de apontar defeitos em classes modificadas pelos mantenedores do sistema para corrigir as falhas catalogadas no repositório iBugs.

Versões Analisadas	**Rhino**		**Ajc**	
	Qtd	%	Qtd	%
Nenhum defeito detectado em classes modificadas	8	42	212	82
Pelo menos um defeito detectado em classes modificadas	11	58	48	18
Total	19	100	260	100

Tabela 3.2: Detecção de defeitos em classes modificadas para correção de falhas

As Figuras 3.10 e 3.11 apresentam informações detalhadas sobre os resultados obtidos. Nessas figuras, o eixo x contém o ID das versões com pelo menos um defeito detectado em classes modificadas. O eixo y indica dois resultados: o número de defeitos apontados pela ferramenta FindBugs em tais classes e, dentre esses defeitos, aqueles que ocorreram em métodos que foram efetivamente alterados na versão após a correção da falha. A Figura 3.10 apresenta, por exemplo, dois defeitos apontados pelo FindBugs nas classes modificadas para correção da falha 114491, sendo que nenhum dos métodos onde os defeitos ocorreram foram efetivamente modificados para a correção dessa falha. Nessa mesma figura, observa-se que a versão 210682 apresenta também dois defeitos apontados pelo FindBugs em classes modificadas para correção da falha, sendo que um método onde localiza-se um desses defeitos foi efetivamente alterado para a correção da falha.

Análise dos Resultados: Analisando a Tabela 3.2 e as Figuras 3.10 e 3.11, conclui-se

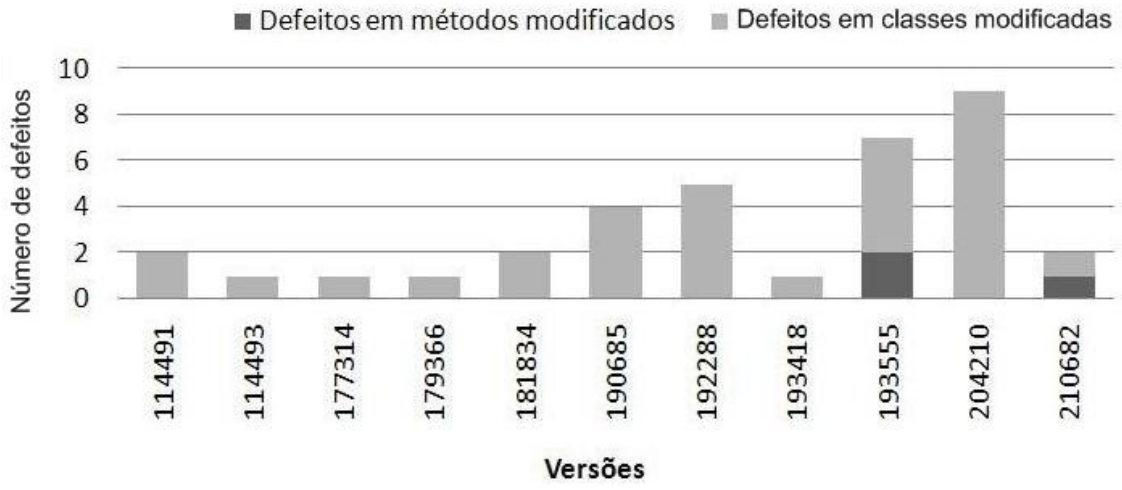

Figura 3.10: Defeitos associados a falhas do interpretador Rhino

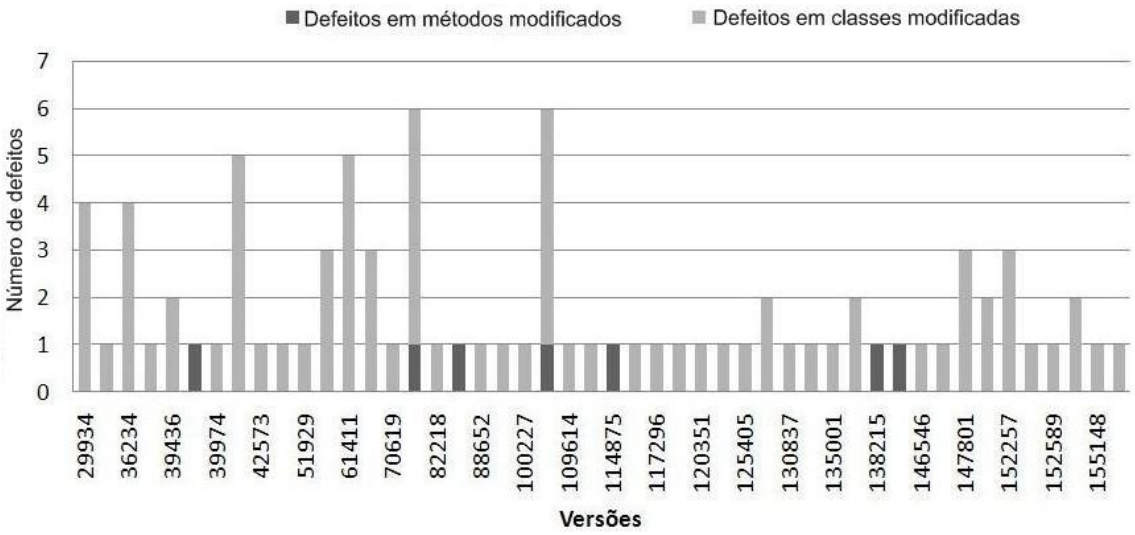

Figura 3.11: Defeitos associados a falhas do compilador Ajc

que a ferramenta FindBugs teria sido pouco útil para detectar elementos de código defeituosos responsáveis pelas falhas reportadas para os dois sistemas analisados. Em primeiro lugar, o número de versões onde a ferramenta não encontrou nenhum defeito em classes modificadas foi significativo (42% das versões do Rhino e 82% das versões do compilador Ajc). Em segundo lugar, mesmo quando a ferramenta FindBugs apontou defeitos em classes modificadas, eles via de regra não ocorreram em métodos alterados pelos mantenedores a fim de corrigir a falha. Por exemplo, em nove das 11 versões analisadas do Rhino não houve nenhuma coincidência entre defeitos apontados pelo FindBugs e métodos alterados na correção da falha. No caso do Ajc, a mesma situação foi observada em 41 das 48 versões analisadas.

Resposta para Questão Q1: Com base nesses resultados, considera-se que a resposta para a questão Q1 é negativa: o FindBugs não teria ajudado a detectar defeitos que deram origem a falhas reportadas por usuários dos sistemas Rhino e Ajc.

3.4 Questão Q2: Ferramentas para Deteção de Defeitos Encontram Defeitos Removidos em Versões Futuras?

O objetivo dessa segunda questão é determinar se os *bugs* de alta prioridade reportados pelo FindBugs em uma versão i dos sistemas considerados no trabalho foram posteriormente removidos pelos mantenedores desses sistemas em uma versão porterior a i. Para responder a essa pergunta, foram consideradas as últimas versões estáveis dos sistemas Rhino e Ajc. Desta forma, foram utilizadas quatorze versões estáveis (*stable releases*) do interpretador Rhino e sete versões estáveis do compilador Ajc. As versões utilizadas com suas respectivas datas de publicação são apresentadas nas Tabelas 3.3 e 3.4.

Rhino	
Versão	**Data**
1.71	06/03/2008
1.67	20/08/2007
1.66	30/07/2007
1.65	19/11/2006
1.64	10/09/2006
1.63	24/07/2006
1.62	19/09/2005
1.61	29/11/2004
1.55	25/03/2004
1.54	10/02/2003
1.53	27/01/2002
1.52	27/07/2001
1.51	10/09/2000
1.4	10/05/1999

Tabela 3.3: Versões do Rhino analisadas

Para automatizar o rastreamento dos defeitos entre essas versões, considerou-se que um defeito reportado no método m de uma classe C em uma versão i foi corrigido em uma versão $i+1$ quando esse mesmo defeito deixa de existir na classe C, método m dessa

Ajc	
Versão	**Data**
1.6.3	23/12/2008
1.6.2	03/10/2008
1.6.1	03/07/2008
1.6.0	23/04/2008
1.6.0rc1	16/04/2008
1.6.0m2	26/02/2008
1.6.0m1	16/01/2008

Tabela 3.4: Versões do Ajc analisadas

nova versão.

Um script em Perl foi desenvolvido para automatizar o cálculo do tempo de vida dos defeitos. O script recebe como entrada o diretório contendo as versões a serem avaliadas. Então, o script executa o FindBugs sobre cada versão. Em seguida, os relatórios de *bugs* resultantes de cada versão analisada são processados. Para cada versão do sistema analisado é gerado um arquivo contendo uma assinatura para cada *bug*. A assinatura de um *bug* foi definida como uma quádrupla contendo as seguinte informações: descrição do *bug*, pacote, classe e método ou atributo. Finalmente os arquivos com as assinaturas são processados para calcular o tempo de vida de cada *bug*, ou seja, se um *bug* de uma versão i, representado por sua assinatura, não ocorreu novamente em uma versão $i+1$, então ele foi corrigido.

3.4.1 Defeitos Avaliados

Defeitos que deixaram de existir em uma versão $i+1$ devido a uma refatoração de código, como por exemplo um método renomeado, não foram considerados no experimento. Desta forma, somente foram considerados defeitos em classes e métodos existentes em todas as versões avaliadas. Por exemplo, suponha um *bug* localizado em um metodo m, classe c e pacote p da versão 1.4 do Rhino. Então somente serão considerados no estudo *bugs* cujos componentes (m;c;p) existam nas versões 1.51 a 1.71. Também não foram considerados no estudo, defeitos detectados em pacotes externos aos programas avaliados. Para os sistemas avaliados, somente o Ajc fazia referência ao pacote externo `org.eclipse` e portanto defeitos detectados nesse pacote foram desconsiderados.

A Tabela 3.5 apresenta o número total de *bugs* reportados pelo FindBugs para o sistema Rhino e o número total de *bugs* considerados no estudo. Como pode ser observado, nesse sistema não houve *bugs* em pacotes externos e portanto somente *bugs* renomeados foram desprezados. Os *bugs* considerados no estudo cobriram de 61% a 100% dos *bugs*

reportados pelo FindBugs.

Versão	**Total (A)**	**Renomeados(B)**	$C = A - B$	C/A
1.4	6	1	5	83,33%
1.51	12	3	9	75,00%
1.52	13	5	8	61,54%
1.53	13	4	9	69,23%
1.54	13	3	10	76,92%
1.55	18	3	15	83,33%
1.61	15	1	14	93,33%
1.62	16	0	16	100,00%
1.63	13	0	13	100,00%
1.64	13	0	13	100,00%
1.65	13	0	13	100,00%
1.66	34	0	34	100,00%
1.67	34	0	34	100,00%
1.71	44	0	44	100,00%

Tabela 3.5: Total de *bugs* reportados pelo FindBugs no Rhino (coluna A), total de *bugs* em elementos renomeados no programa (coluna B) e total de *bugs* considerados no estudo (coluna C)

Da mesma forma, a Tabela 3.6 apresenta o número total de *bugs* reportados pelo FindBugs e o número total de *bugs* considerados no estudo para o Ajc. Nesse caso, o estudo cobriu de 24% a 31% do total de *bugs* reportados pelo FindBugs. A maioria do *bugs* desconsiderados no estudo foram detectados no pacote `org.eclise` externo ao Ajc.

Versão	**Total (A)**	**Externos (B)**	**Renomeados (C)**	$D = A - B - C$	D/A
160m1	315	211	17	87	27,62%
160m2	318	211	17	90	28,30%
160rc1	318	211	17	90	28,30%
160	318	211	17	90	28,30%
161	325	211	12	102	31,38%
162	285	212	2	71	24,91%
163	298	212	0	86	28,86%

Tabela 3.6: Total de *bugs* reportados pelo FindBugs no Ajc (coluna A), total de *bugs* em pacotes externos (coluna B), total de *bugs* em elementos renomeados no programa (coluna C) e número total de *bugs* considerados no estudo (coluna D)

3.4.2 Resultados

As Tabelas 3.7 e 3.8 apresentam os resultados obtidos nesta análise. Para cada versão i dos sistemas analisados, as tabelas mostram o número de defeitos apontados pelo FindBugs nessa versão. Dentre tais defeitos, as tabelas mostram ainda quantos defeitos continuaram presentes nas versões subsequentes a i. Por exemplo, os cinco defeitos apontados pelo FindBugs na versão 1.4 do Rhino não foram corrigidos na versão seguinte (versão 1.51). No entanto, um destes defeitos foi corrigido na versão 1.52, restando, portanto quatro defeitos nessa versão. Esses mesmos quatro defeitos continuaram presentes na versões 1.53 e 1.54. Por fim, na última versão analisada (versão 1.7), restou um defeito dentre os cinco defeitos inicialmente detectados. Resumindo, ao longo das versões seguintes, os mantenedores do Rhino corrigiram quatro dentre os cinco defeitos apontados pelo FindBugs na versão inicial.

A mesma análise foi repetida assumindo como inicial cada uma das versões estáveis do interpretador Rhino. Por exemplo, na versão 1.51, o FindBugs apontou nove defeitos, sendo que cinco deles surgiram na versão 1.4 (e como não foram removidos, se manifestaram novamente na versão 1.51). Como a análise realizada procura por defeitos surgidos inicialmente em uma versão i nas versões subsequentes a i, não existem defeitos abaixo da diagonal principal das Tabelas 3.7 e 3.8.

Origem dos defeitos	**Quantidade de defeitos Pendentes**													
	1.4	1.51	1.52	1.53	1.54	1.55	1.61	1.62	1.63	1.64	1.65	1.66	1.67	1.7
1.4	5	5	4	4	4	3	3	3	3	3	3	1	1	1
1.51		4	2	2	2	2	2	2	2	2	2	2	2	1
1.52			2	0	2	2	2	2	2	2	2	0	0	0
1.53				2	0	0	0	0	0	0	0	0	0	0
1.54					1	0	0	0	0	0	0	0	0	0
1.55						8	4	3	3	3	3	3	3	3
1.61							3	1	1	1	1	1	1	1
1.62								5	2	2	2	2	2	2
1.63									0	0	0	0	0	0
1.64										0	0	0	0	0
1.65											0	0	0	0
1.66												25	25	25
1.67													0	0
1.71														11
Total	5	9	8	8	9	15	14	16	13	13	13	34	34	44

Tabela 3.7: Tempo de vida de defeitos no interpretador Rhino

Exemplo de Defeito Removido: Conforme apresentado na Tabela 3.7, dentre os quatro defeitos inicialmente detectados na versão 1.51, houve a correção de dois defeitos na versão 1.52. Um desses defeitos é mostrado nas linhas 5 e 10 da Figura 3.12. Ele foi detectado

Origem dos defeitos	Quantidade de defeitos Pendentes						
	1.6.0m1	1.6.0m2	1.6.0rc1	1.6.0	1.6.1	1.6.2	1.6.3
1.6.0m1	87	87	86	86	81	56	56
1.6.0m2		3	3	3	3	1	1
1.6.0rc1			1	1	1	1	1
1.6.0				0	0	0	0
1.6.1					17	12	12
1.6.2						1	1
1.6.3							15
Total	87	90	90	90	102	71	86

Tabela 3.8: Tempo de vida de defeitos no compilador Ajc

pelo *bug pattern* RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE da categoria corretude. Esse detector procura por situações em que uma referência é testada se igual a *null* (linha 10), porém, em um ponto anterior ao teste, o mesmo objeto por ela referenciado foi acessado (linha 5). De acordo com a documentação desse *bug pattern*, esses dois pontos de código discordam entre si quanto a possibilidade de o objeto conter o valor *null*, o que significa que, ou o objeto não pode conter o valor *null* e portanto o teste é redundante ou a referência realizada antes do teste é um erro que pode resultar em *null pointer exception*. A Figura 3.13 apresenta o código com a correção realizada na versão 1.52, onde observa-se que o teste para verificar se a referência `thisObj` possui valor diferente de *null* é realizado na linha 5, antes de qualquer acesso ao objeto por ela referenciado.

```
 1: public final class OptRuntime extends ScriptRuntime {
 2:   public static Object thisGet(Scriptable thisObj, String id,
 3:                                Scriptable scope)
 4:   {
 5:       Object result = thisObj.get(id, thisObj);
 6:       if (result != Scriptable.NOT_FOUND)
 7:           return result;
 8:
 9:       Scriptable start = thisObj;
10:       if (start == null) {
11:            throw Context.reportRuntimeError(
12:                getMessage("msg.null.to.object", null));
13:        }
14:        ...
15:    }
16: }
```

Figura 3.12: Exemplo de *Bug* detectado na versão 1.51 do Rhino

```
 1: public final class OptRuntime extends ScriptRuntime {
 2:   public static Object thisGet(Scriptable thisObj, String id,
 3:                                Scriptable scope)
 4:   {
 5:       if (thisObj == null) {
 6:           throw Context.reportRuntimeError(
 7:               getMessage("msg.null.to.object", null));
 8:       }
 9:
10:       Object result = thisObj.get(id, thisObj);
11:       if (result != Scriptable.NOT_FOUND)
12:            return result;
13:       ...
14:    }
15: }
```

Figura 3.13: Exemplo de correção de *bug* realizado na versão 1.52 do Rhino

Análise dos Resultados: Os resultados das Tabelas 3.7 e 3.8 mostram que o número de defeitos corrigidos ao longo das versões analisadas do interpretador Rhino e Ajc foram percentualmente semelhantes. A taxa de correção de defeitos para os dois sistemas avaliados foi da ordem de 30% (34% para o Rhino e 31% para o Ajc). Dos 66 defeitos únicos identificados para o interpretador Rhino (os quais correspondem ao somatório dos valores da diagonal principal da Tabela 3.7), 44 defeitos continuavam presentes na última versão analisada (isto é, 66%). Para o compilador Ajc , dos 124 defeitos únicos reportados ao longo das versões analisadas, 86 defeitos (ou 69%) continuavam presentes na última versão analisada. Além disso, é importante mencionar que o número de defeitos encontrados no Ajc foi praticamente duas vezes maior que o número de defeitos encontrados no Rhino (124 defeitos, contra 66 defeitos) sendo que o Ajc é 1.5 vezes maior que o tamanho do Rhino (75 KLOC, contra 49 KLOC).

Resposta para Questão Q2: Desta forma, podemos concluir que a resposta para a questão Q2 foi parcialmente positiva: cerca de 30% dos defeitos apontados pela ferramenta FindBugs para os sistemas Rhino e Ajc foram de fato removidos em versões futuras desse sistema.

3.5 Riscos à Validade do Estudo de Caso

Nesta seção, os resultados e as conclusões do estudo descrito neste capítulo são avaliados segundo sua validade interna, externa e de construção [PPV97].

Validade Externa: Essa forma de validade se refere ao grau de aplicabilidade das conclusões de um estudo a uma população mais ampla. O estudo descrito neste capítulo envolveu dois sistemas (Rhino e Ajc) e uma única ferramenta para detecção de defeitos (FindBugs), o que a princípio poderia comprometer a generalização de suas conclusões. No entanto, esse risco é atenuado pelos seguintes motivos:

1. Os sistemas utilizados no estudo são de médio/grande porte e de relativa complexidade. Além dissso, esses sistemas possuem um histórico bem documentado de falhas reportadas por usuários finais. Também utilizou-se somente defeitos classificados como falhas e defeitos de alta prioridade.

2. O FindBugs é uma das ferramentas para detecção de defeitos mais completas e populares em se tratando de sistemas implementados em Java. Portanto, acredita-se que os resultados obtidos no trabalho são representativos e comparáveis àqueles que poderiam ser obtidos por meio de outras ferramentas para detecção de defeitos disponíveis para essa linguagem. No entanto, os mesmos não devem ser generalizados para sistemas implementados em outras linguagens, como C e C++.

Validade Interna: Essa forma de validade avalia se as conclusões obtidas não são resultantes de fatores que não foram controlados ou medidos. No planejamento do estudo, uma grande preocupação foi confirmar se a ferramenta FindBugs já não teria sido usada pelos próprios desenvolvedores das versões analisadas do Rhino e Ajc. Para esclarecer essa dúvida, uma mensagem foi enviada ao fórum oficial de desenvolvedores desses sistemas, que confirmaram que o FindBugs não é usado durante o processo de *building* dos mesmos.

Validade de Construção: Essa forma de validade procura avaliar se as conclusões obtidas não são resultantes de uma condução incorreta do experimento (por exemplo, devido a dados incorretos que foram gerados). Conforme afirmado na Seção 3.2, a primeira parte do experimento realizado considerou somente os *bugs* do repositório classificados como falha, visto que a literatura pesquisada menciona que somente essa categoria de defeitos pode ser detectada por ferramentas para detecção de defeitos. Na segunda parte do estudo, foram considerados apenas *bugs* de alta prioridade reportados pelo FindBugs. Ou seja, considerou-se somente *bugs* com maior probabilidade de serem verdadeiros positivos.

Além disso, para responder à questão Q2, foram desconsiderados os defeitos removidos em função de refatoração de código e defeitos detectados em pacotes externos aos sistemas avaliados.

3.6 Conclusões

As principais conclusões da experiência descrita neste capítulo são as seguintes:

- Ferramentas para detecção de defeitos não são efetivas para apontar defeitos associados a falhas reportadas por usuários finais. Ou seja, um mantenedor de software encarregado de corrigir uma falha apontada por um usuário final não deve investir tempo em usar essas ferramentas para ajudar a detectar o elemento de software defeituoso.

- Em alguma medida, ferramentas para detecção de defeitos são capazes de apontar defeitos que, mais cedo ou mais tarde, desenvolvedores deverão corrigir em versões futuras de um sistema. Ou seja, ao se antecipar e corrigir imediatamente os defeitos apontados por essas ferramentas, desenvolvedores podem estar contribuindo para aumentar a qualidade dos sistemas sob sua responsabilidade.

Adicionalmente, durante o uso dessas ferramentas os seguintes problemas foram identificados:

1. Grande número de defeitos detectados: Mesmo configurando a ferramenta para reportar somente defeitos de alta prioridade, o número de defeitos gerados foi alto (por exemplo, 315 defeitos no Ajc).

2. Eficiência do critério de priorização de defeitos: O critério de priorização de defeitos dessas ferramentas nem sempre é eficiente ou adequado a um determinado sistema. Por exemplo, se um sistema tem como objetivo servir como compilador ou interpretador, então defeitos relacionados a segurança podem ser considerados de baixa prioridade.

3. Dificuldade para analisar e manter relatórios de defeitos: Normalmente, essas ferramentas reportam os erros em uma interface gráfica ou em arquivo XML. O trabalho de análise de defeitos por uma equipe de qualidade contendo duas ou mais pessoas, exigiria um esforço para carga dos resultados em um banco de dados bem como uma ferramenta que possar permitir tarefas de gerenciamento, produção de estatísticas e relatórios.

4. Dificuldade para uso de duas ou mais ferramentas para detecção de defeitos: O uso de duas ou mais ferramentas para detecção de defeitos poderia aumentar a taxa de verdadeiros positivos localizados, contribuindo para uma melhor qualidade do produto final. No entanto, os relatórios gerados por essas ferramentas são totalmente

distintos, sendo necessário o emprego de alguma técnica para consolidação de resultados, removendo por exemplo, defeitos detectados por mais de uma ferramenta.

Os problemas acima reportados podem desmotivar o uso de ferramentas para detecção de defeitos no processo de desenvolvimento de sistemas. Desta forma, foi desenvolvida uma ferramenta para contornar essas limitações. Esta ferramenta, a ser apresentada no próximo capítulo, tem como objetivo principal atuar como interface de execução de uma ou mais ferramentas para detecção de defeitos. Por exemplo, ela elimina duplicidade de defeitos entre as ferramentas e permite a composição de *bug patterns* com um critério de priorização mais flexível.

Capítulo 4

A Ferramenta Smart Bug Detector

4.1 Visão Geral

Após as experiências de avaliação e utilização de ferramentas para detecção de defeitos, descritas nos Capítulos 2 e 3, este capítulo apresenta uma proposta de solução para uso de uma ou mais ferramentas para detecção de defeitos. De forma resumida, descreve-se o desenvolvimento de uma ferramenta que funciona como uma interface entre o usuário e diversas ferramentas para detecção de defeitos. Mais especificamente, a meta-ferramenta para deteção de defeitos descrita neste capítulo, chamada Smart Bug Detector, se propõe a resolver ou minimizar os seguintes problemas:

- Dificuldade para uso de duas ou mais ferramentas para detecção de defeitos: O uso de duas ou mais ferramentas para detecção de defeitos pode contribuir para aumentar a taxa de verdadeiros positivos localizados [WAWS08]. No entanto, o uso combinado de duas ou mais ferramentas para detecção de defeitos é complexo devido ao esforço necessário para configurar e executar diversas ferramentas e analisar o resultado gerado por cada uma delas. Assim, a solução apresentada tem como objetivo principal permitir o uso de duas ou mais ferramentas, detectando defeitos duplicados e consolidando todos os relatórios gerados em um formato único armazenado em uma tabela em banco de dados.

- Grande número de defeitos reportados: Ferramentas para deteção de defeitos geram um grande número de defeitos e normalmente o resultado da deteção é gravado em um arquivo XML ou exibido diretamente na interface da ferramenta, o que dificulta o trabalho de análise por uma equipe de qualidade ou grupo de desenvolvedores. Como solução, propõe-se que o resultado seja armazenado em banco de dados proporcionando o compartilhamento do resultado de um processo de deteção de defeitos

entre os membros do projeto.

- Eficiência do critério de priorização de defeitos: O critério de priorização de defeitos das ferramentas para detecção de defeitos nem sempre é eficiente ou adequado a um determinado sistema [KE07]. Como solução propõe-se uma funcionalidade que permite alterar as prioridades dos *bug patterns* das ferramentas para detecção de defeitos utilizadas.

Considerando os problemas observados, as principais funcionalidade da ferramenta Smart Bug Detector são as seguintes:

- Execução de uma ou mais ferramentas para detecção de defeitos: A solução proposta permite executar uma ou mais ferramentas para detecção de defeitos para inspecionar um projeto de software aumentando assim a capacidade de detecção de *bugs* sem que o usuário tenha que lidar com todos os detalhes de uso inerente a cada ferramenta.

- Composição de defeitos: A solução proposta permite criar novos *bug patterns* por meio de composição com os *bug patterns* extraídos das ferramentas para detecção de defeitos. Dessa forma, o usuário da solução pode definir novos detectores sem a necessidade de implementar código que acesse a API das ferramentas para detecção de defeitos utilizadas.

- Rastreamento de defeitos entre versões: Os defeitos resultantes são armazenados em uma tabela de banco de dados uma única vez, independente da versão do sistema analisado. Devido ao grande número de defeitos detectados, essa funcionalidade é essencial para preservar o trabalho de análise e classificação de defeitos ao longo das versões de um sistema.

- Flexibilidade para definição de prioridades e classes de defeitos: A solução proposta permite que se cadastrem prioridades e classes de defeitos. As prioridades cadastradas são utilizadas na funcionalidade de composição de defeitos, ou seja, defeitos detectados através de uma composição terão a prioridade definida pelo usuário da aplicação. Além disso, as classes de defeitos cadastradas são utilizadas na funcionalidade de classificação de defeitos detectados. A classificação de defeitos é uma atividade realizada pelo usuário da aplicação após a execução de uma ferramenta para detecção de defeitos. Nessa atividade, o usuário analisa a descrição de um *bug*, avalia características como, por exemplo, prioridade e código onde o *bug* foi detectado e finalmente classifica o *bug* em uma determinada classe como, por exemplo, falso positivo, trivial, sério, etc.

- Classificação de defeitos de forma compartilhada: Após a detecção de defeitos em um sistema e apresentação dos resultados, a solução possibilita o uso compartilhado, facilitando a divisão do trabalho de análise e classificação de defeitos reportados.

- Exibição de métricas: A solução proposta disponibiliza métricas sobre detecções de defeitos realizadas, permitindo por exemplo identificar classes ou módulos mais propensos a erros. As métricas disponibilizadas pela implementação atual da ferramenta são as seguintes: quantidade de defeitos por classe e quantidade de defeitos por *bug pattern*.

De forma a validar a solução aqui proposta, desenvolveu-se um protótipo utilizando a linguagem `C#` e o banco de dados SQL Server 2008. A ferramenta implementada foi chamada de Smart Bug Detector.

A Figura 4.1 apresenta a tela principal da solução, cujo *layout* foi inspirado na interface do FindBugs. O painel da esquerda exibe em uma árvore os defeitos detectados ordenados por critérios como classe, *bug pattern* e descrição do *bug*. Os critérios de ordenação existentes no protótipo criado permitem exibir os *bugs* em ordem de classe e prioridade ou prioridade e classe. O painel da direita apresenta, para cada defeito selecionado, o código fonte da classe onde o defeito foi detectado – desde que o código fonte do sistema esteja disponível. O painel inferior, por sua vez, apresenta mais informações sobre o defeito selecionado. Essas informações incluem a descrição do defeito, a classe, método e atributo onde o defeito foi detectado, a classificação do defeito (para defeitos já classificados) e uma URL que possibilita a visualização de uma eventual página web com mais detalhes e exemplos sobre o defeito. Normalmente, os autores das ferramentas para detecção de defeitos disponibilizam um *web site* com documentação e exemplos sobre os *bug patterns* de suas ferramentas[1].

4.2 Funcionalidades Básicas

4.2.1 Pré-requisitos

Para utilização da solução implementada neste trabalho de dissertação, três condições devem ser atendidas:

- Configuração das ferramentas para detecção de defeitos a serem utilizadas pela interface: Cada ferramenta para detecção de defeitos a ser utilizada deve ser instalada

[1]Para o FindBugs, essas descrições estão disponíveis em http://findbugs.sourceforge.net/bugDescriptions.html. Para o PMD, as descrições estão disponíveis em http://pmd.sourceforge.net/rules/

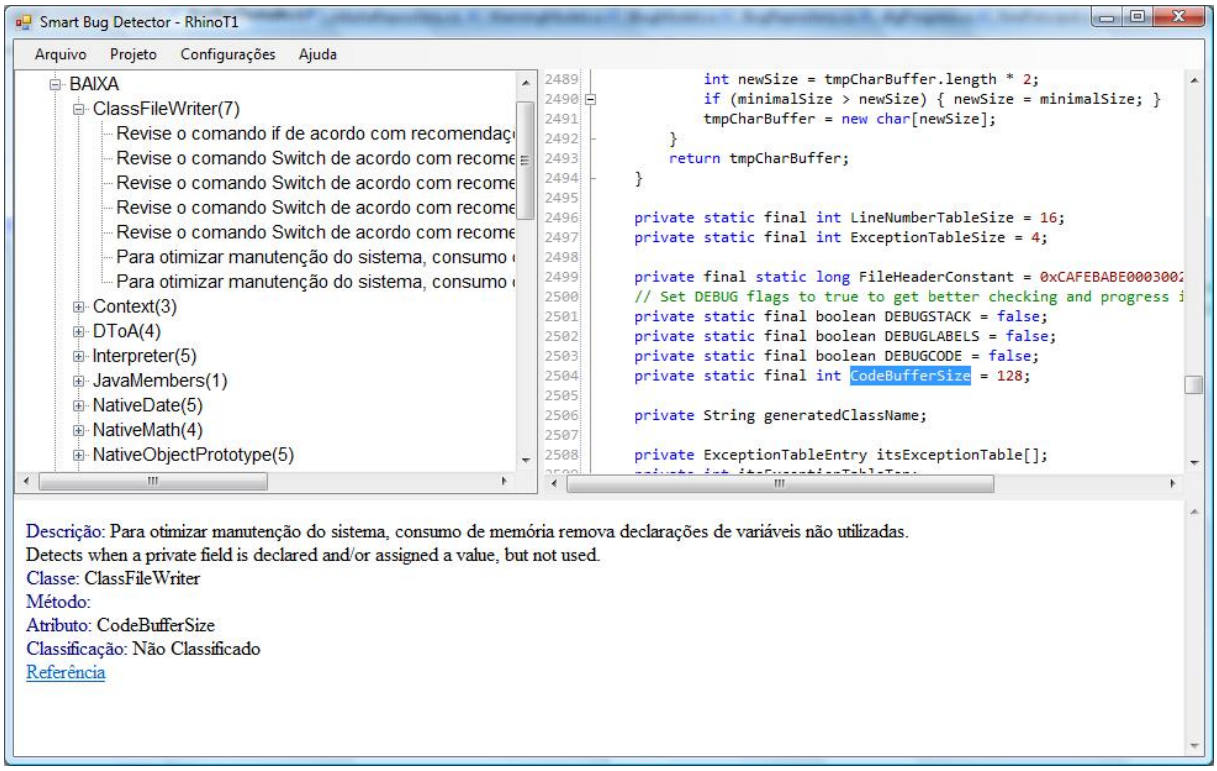

Figura 4.1: Visão geral da interface proposta

e testada isoladamente no mesmo computador de execução da interface.

- Banco de dados SQL Server: A solução proposta utiliza um banco de dados SQL Server para armazenamento de informações como *bugs* detectados, cadastros de ferramentas para detecção de defeitos e seus *bug patterns* etc. Juntamente com a instalação da ferramenta Smart Bug Detector, disponibiliza-se um *script* para criação do banco de dados utilizado pela aplicação.

- Configuração da conexão com o banco de dados: Após a instalação da interface, deve-se editar a chave DB_ANALISE no arquivo `SmartBugDetector.exe.config` informando o servidor SQL no campo `Data Source`, o banco de dados no campo `Initial Catalog` e usuário e senha de conexão, conforme exemplificado na Figura 4.2.

4.2.2 Configuração de Ferramentas para Detecção de Defeitos

Com os pré-requisitos atendidos, o primeiro passo para uso da interface é realizar a configuração das ferramentas para detecção de defeitos a serem utilizadas. A Figura 4.3 apresenta a tela de configuração com as funcionalidades de salvar, editar e excluir cadastro

```
1: <configuration>
2:  <connectionStrings>
3:    <add name="DB_ANALISE"
      connectionString="Data Source=SILVIO-NOTE\SQLSILVIO;
      Initial Catalog=DB_ANALISE;User ID=sa;pwd=sa"/>
4:  </connectionStrings>
5: </configuration>
```

Figura 4.2: Exemplo de configuração de conexão da aplicação com banco de dados

de ferramentas e importação de *bug patterns*. Para cada ferramenta para detecção de defeitos que se deseja utilizar, deve ser cadastrado um registro informando o nome da ferramenta e o diretório de instalação da mesma.

A função de importação de *bug patterns* realiza a tarefa de extrair os *bug patterns* de cada ferramenta. Os dados obtidos por esse processo são armazenados em uma tabela e posteriormente poderão ser utilizados na composição de novos *bugs patterns*.

Como as ferramentas para detecção de defeitos não seguem um mesmo padrão para descrição de *bug patterns*, a solução criada não é capaz de reconhecer e utilizar qualquer ferramenta. O protótipo implementado inclui suporte ao FindBugs e PMD. No entanto, a solução pode ser adaptada para uso de outras ferramentas.

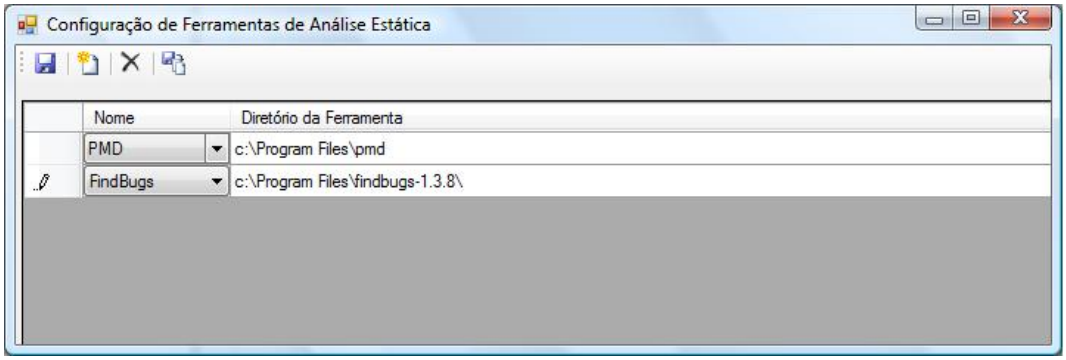

Figura 4.3: Configuração das ferramentas para detecção de defeitos

4.2.3 Cadastro de Prioridades e Classes de Defeitos

Duas das principais funcionalidades da ferramenta Smart Bug Detector, denominadas composição de *bug patterns* e classificação de defeitos, requerem que seja realizado um cadastro prévio de prioridades e classes de defeitos, conforme descrito a seguir:

- Cadastro de prioridades: Esse cadastro permite que novas prioridades sejam adici-

onadas na ferramenta, as quais poderão ser posteriormente associadas a novos *bug patterns*. Todo *bug pattern* cadastrado na ferramenta tem obrigatoriamente uma prioridade. Os *bugs* detectados através da aplicação de uma composição recebem a prioridade atribuida à composição. Os *bugs* detectados sem a aplicação de uma composição recebem a prioridade de maior peso cadastrada, considerada a prioridade de menor relevância. A Figura 4.4 apresenta a tela utilizada no cadastro de prioridades, onde informa-se o nome da prioridade e o peso a ela atribuído.

Figura 4.4: Cadastro de prioridades de *bug patterns*

- Cadastro de classes de defeitos: Os registros criados através deste cadastro são utilizados para classificar defeitos detectados. A Figura 4.5 apresenta a tela utilizada nesse cadastro, enquanto que a Figura 4.6 apresenta o uso desta funcionalidade. Nessa última figura, o usuário aciona o botão direito do mouse sobre um *bug* na árvore de *bugs* e o sistema exibe um *menu* com as opções de classificação de defeitos. Ao selecionar a opção de classificação desejada o sistema atualiza o registro do *bug* com a classificação selecionada.

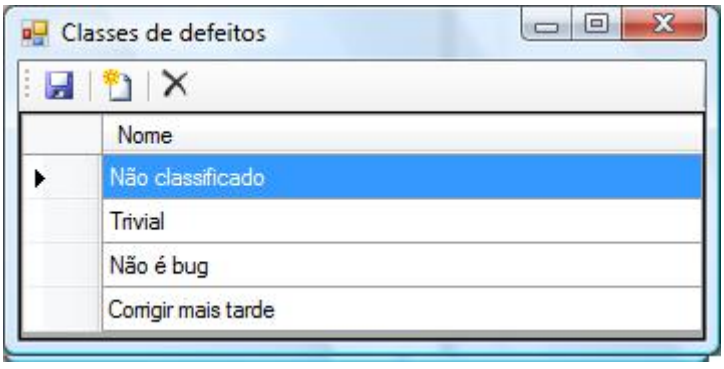

Figura 4.5: Cadastro de classes de defeitos

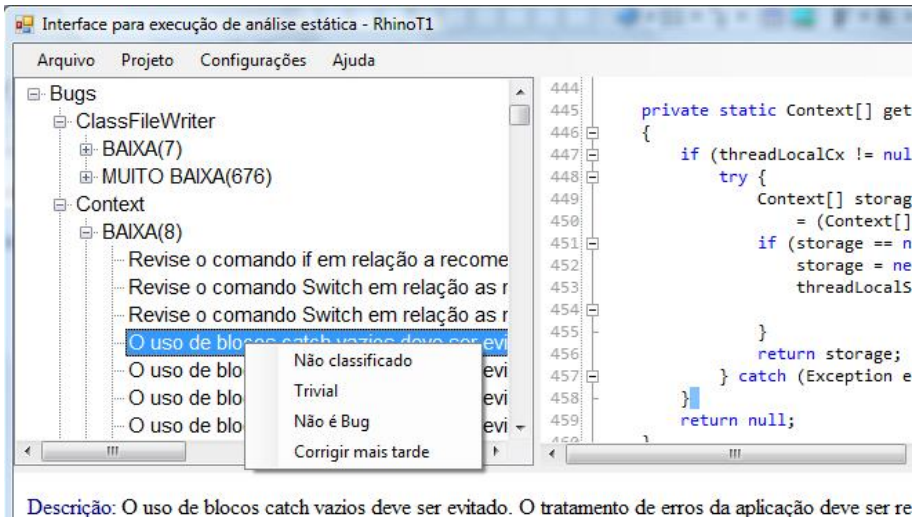

Figura 4.6: Classificação de defeitos

4.2.4 Composição de *Bug Patterns*

A funcionalidade de composição de *bug patterns* permite que novas regras de detecção de *bugs* sejam criadas a partir de detectores oferecidos pelas ferramentas para detecção de defeitos. Os tipos de composição de *bug patterns* possíveis de serem criados são os seguintes:

- `Alias`: Esse tipo de composição permite redefinir um *bug pattern* implementado por ferramentas para detecção de defeitos, atribuindo-lhe uma nova descrição ou prioridade. O objetivo desse tipo de composição é permitir a tradução de uma mensagem de *bug* ou redefinição de sua prioridade. A Figura 4.7 exemplifica o princípio de funcionamento desse tipo de composição. Nessa figura, observa-se que para todo *bug* detectado pelo *bug pattern* da composição `Alias`, um *bug* R é reportado.

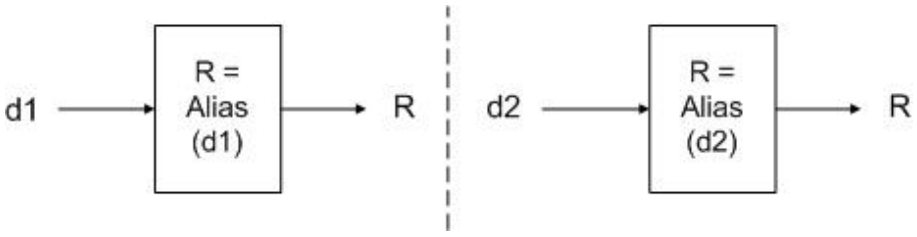

Figura 4.7: Composição `Alias`

- Composição `Or`: Permite combinar *bug patterns* das ferramentas para detecção de defeitos utilizando o operador `Or`. A Figura 4.8 apresenta o princípio de funcionamento dessa composição. Nesse exemplo, observa-se uma composição R formada

pelos *bug patterns* d1 ou d2. Para qualquer combinação de um ou mais *bugs* detectados por d1 ou d2, um *bug R* será reportado.
Dessa forma, esse tipo de composição permite remoção de duplicidade de *bugs* detectados ao se combinar *bug patterns* de diferentes ferramentas para detecção de defeitos.

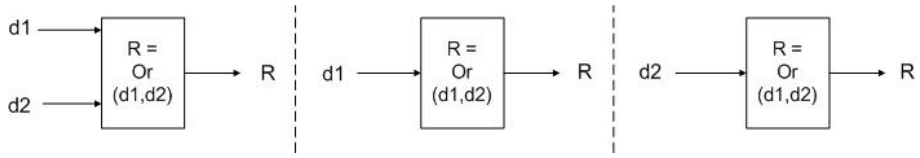

Figura 4.8: Composição `Or`

- Composição `And`: Uma composição `And` permite combinar diversos detectores com o operador lógico `And` sendo aplicado como regra de combinação. Por exemplo, se uma composição R é composta pelos *bug patterns* d1 e d2, um *bug R* será reportado somente se houver *bugs* detectados por d1 e d2 em uma mesma classe ou método, de acordo com o escopo de consolidação selecionado para a composição. O princípio de funcionamento desse tipo de composição é apresentado na Figura 4.9.

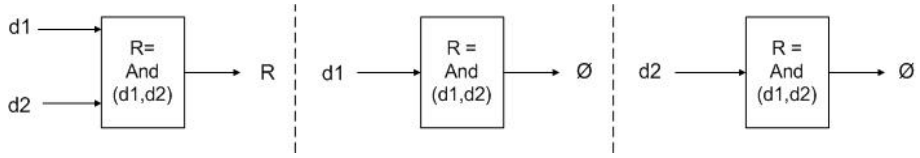

Figura 4.9: Composição `And`

O escopo de consolidação é um atributo indicado no cadastro de uma composição que permite indicar como os *bugs* serão consolidados. Esse atributo se aplica somente às composições do tipo `And` e `Or`. Os valores possíveis são:

- Método: Se uma composição R é composta pelos *bugs patterns* d1 e d2, então um bug *R* será reportado somente se dois *bugs* forem detectados em um mesmo método.

- Classe: Neste caso os *bugs* detectados deverão pertencer à mesma classe para que a composição gere um resultado.

Além do escopo de consolidação, pode-se definir também o escopo de aplicação de uma composição. Essa definição é realizada no cadastro de uma composição por meio do campo de nome `Escopo de Aplicação`. Esse campo deve ser preenchido com o nome de uma

classe ou pacote, quando se deseja que a composição cadastrada seja aplicada somente em um determinado módulo ou classe de um sistema. Por exemplo, se um sistema for organizado em três pacotes `Faturamento.UI`, `Faturamento.Domain` e `Faturamento.DataAcess`, então toda composição cadastrada com o campo *Escopo de Aplicação* contendo o valor `Faturamento.Domain` será aplicada somente em classes desse pacote.

A Figura 4.10 apresenta a tela para cadastro de composições onde devem ser fornecidas as seguintes informações:

- Nome do *bug pattern* - o nome do *bug* a ser exibido na árvore de resultados.
- Descrição: contém a informação a ser apresentada explicando o problema detectado quando se selecionar o *bug* na árvore.
- Prioridade: permite definir a prioridade desejada para os *bug* detectados através da composição.
- Projeto de uso: permite indicar se o *bug pattern* será de uso em qualquer projeto ou somente no projeto corrente.
- Escopo de consolidação: permite indicar se a ferramenta deverá considerar os *bugs* em nível de classe ou método durante a aplicação da composição.
- Escopo de Aplicação: permite indicar a classe ou pacote de aplicação da composição.
- *Bug Patterns* originais: Contém a coleção de *bug patterns* utilizados na composição. O botão adicionar exibe um tela para pesquisa e seleção dos itens desejados na composição. O botão remover exclui da composição os itens selecionados.

4.2.5 Criação e Execução de um Projeto para Detecção de Defeitos

Após a fase de configuração, para utilizar a ferramenta Smart Bug Detector em um sistema qualquer, deve-se criar um projeto. A Figura 4.11 apresenta a tela exibida para criação de um projeto. As principais informações a serem indicadas nessa tela são o nome do projeto, o caminho dos arquivos binários e do código fonte da aplicação a ser analisada e as ferramentas para detecção de defeitos a serem utilizadas. No projeto pode-se ainda configurar filtros e o critério de ordenação para exibição dos *bugs* detectados. As opções de filtros disponíveis permitem a exibição de defeitos por prioridades ou classificações indicadas.

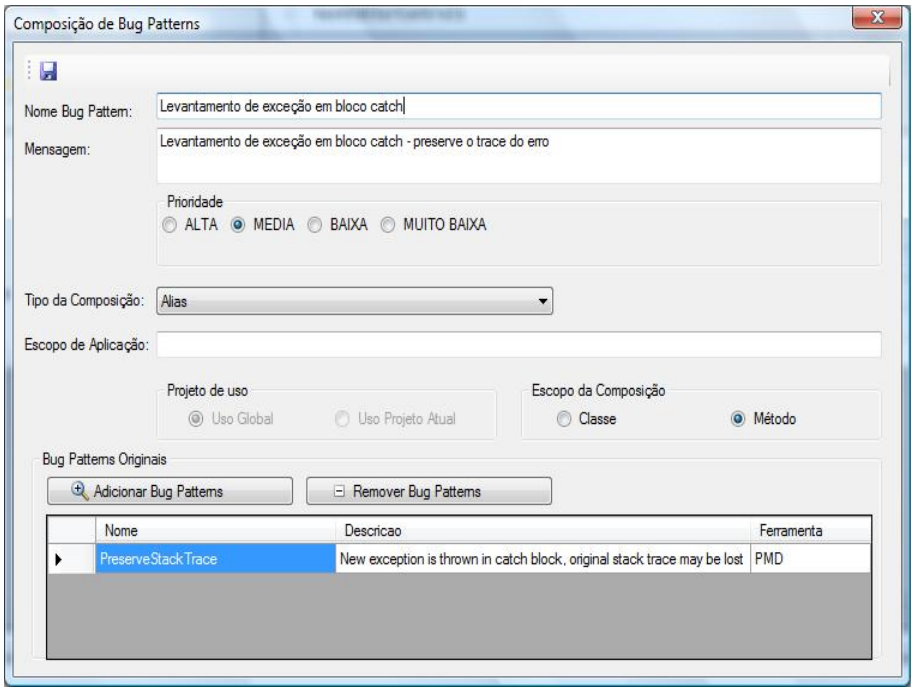

Figura 4.10: Composição de *bug patterns*

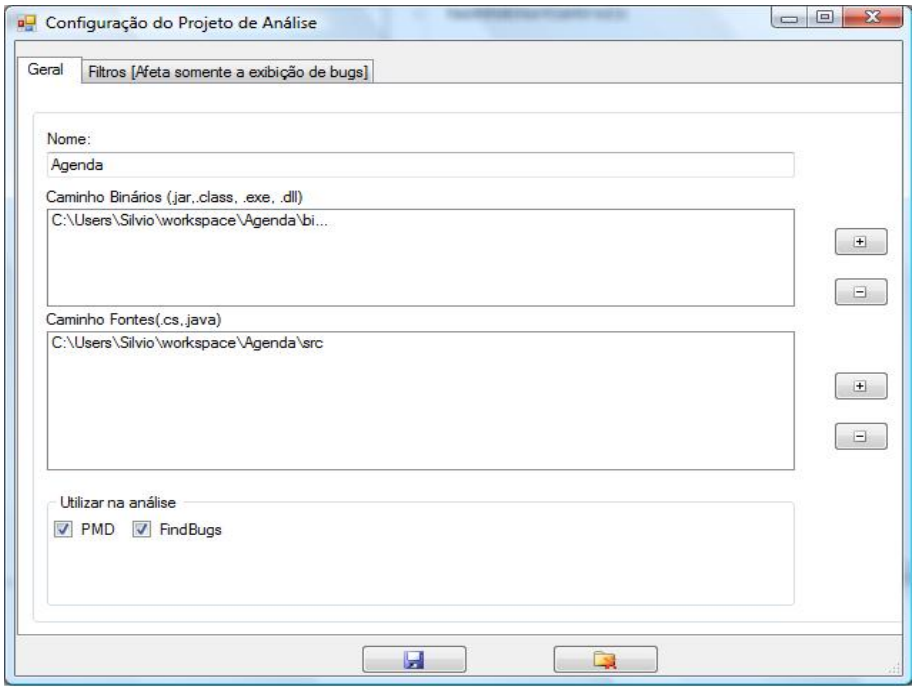

Figura 4.11: Tela para criar ou alterar um projeto de detecção de defeitos

4.2.6 Métricas

Essa funcionalidade tem por objetivo exibir métricas sobre o resultado de uma detecção realizada. No protótipo criado neste trabalho dois tipos de métricas estão disponíveis:

- Total por Bug Pattern - apresenta um gráfico de pizza com o total de *bugs* detectados por tipo de *bug pattern*, permitindo visualizar os tipos de defeitos mais frequentes em um sistema.

- Total por Classe ou Pacote - apresenta um gráfico de pizza com total de *bugs* detectados por classe ou pacote permitindo visualizar classes e módulos mais propensos a defeitos.

4.3 Arquitetura Interna

A arquitetura da aplicação, apresentada na Figura 4.12, foi organizada no padrão de camadas [FRF+02]. As camadas utilizadas no projeto bem como suas responsabilidades são as seguintes:

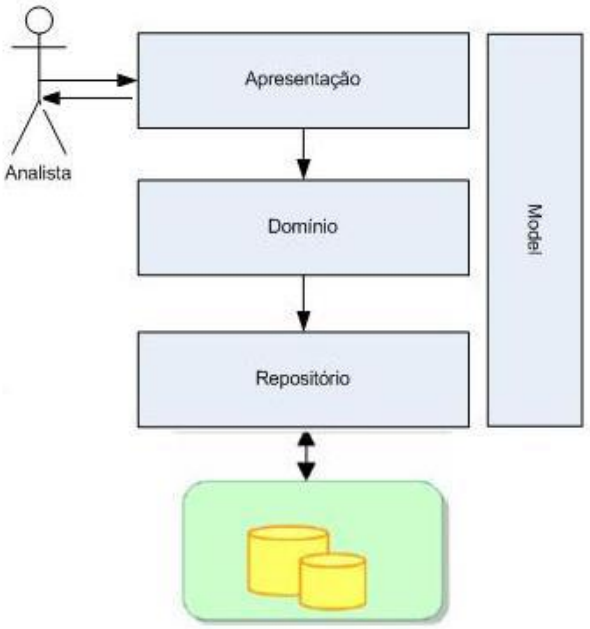

Figura 4.12: Arquitetura da ferramenta Smart Bug Detector

1. Camada de Apresentação: Camada formada por formulários Windows contendo somente a lógica de apresentação de dados e interface com usuário. A Figura 4.13 apresenta o diagrama de classes dessa camada. As principais funcionalidades da aplicação, descritas na Seção 4.2, estão representadas nesse diagrama por suas respectivas classes. Por exemplo, a classe `frmPrincipal` representa o formulário principal da aplicação. Nesse diagrama pode ser observado que as principais classes

de cadastro herdam da classe `frmCadastroBase`. Essa classe foi criada para permitir reutilização por herança de código do *layout* dos formulários. Nessa classe adicionou-se, por exemplo, uma barra de ferramentas com botões para as operações de inserir, editar e excluir. As suas subclasses apenas adicionam código às operações dos botões já existentes.

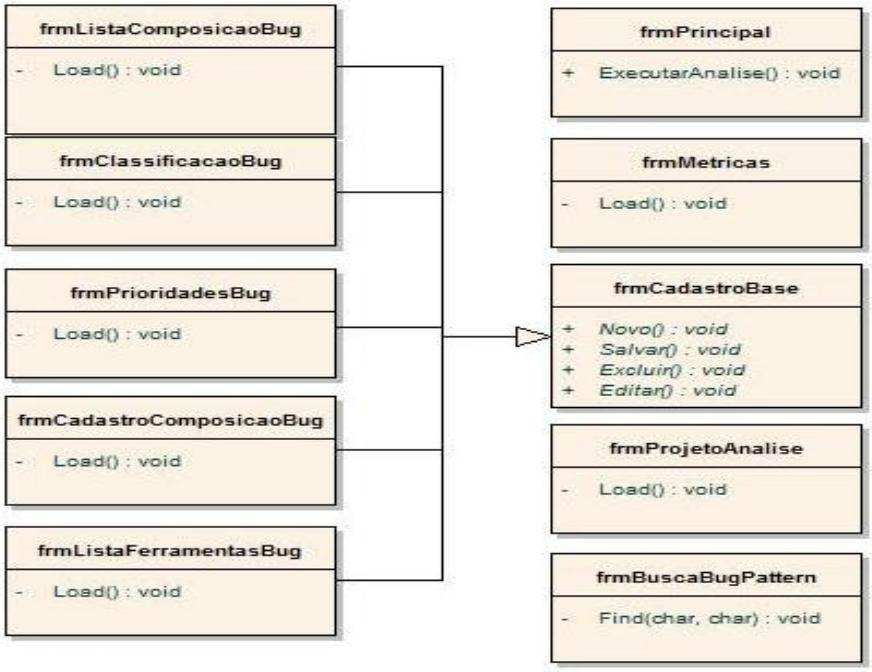

Figura 4.13: Diagrama de classes da camada de apresentação

2. Camada de Domínio: Camada responsável pela lógica da aplicação, ou seja, ela recebe requisições da camada de apresentação e realiza execução de processos, transformações e validações necessárias e eventualmente repassa para a camada de repositório uma solicitação de interação com o banco de dados. A Figura 4.14 apresenta o diagrama de classes dessa camada. Dentre as classes apresentadas nessa figura, cita-se por exemplo, a classe `AnaliseDomain` responsável por executar um processo de detecção de defeitos através de chamada ao método `ExecutarAnalise()`.

3. Camada de Repositório: Camada de persistência responsável pela interação com o banco de dados. As classes dessa camada realizam o papel de persistência e mapeamento entre as tabelas de banco de dados e os objetos da camada de modelo. A Figura 4.15 apresenta o diagrama de classes dessa camada. Uma das classes principais apresentada nesse diagrama é a classe `RepositoryBase` da qual herdam todas as demais classes da camada Repositório. A classe `RepositoryBase`

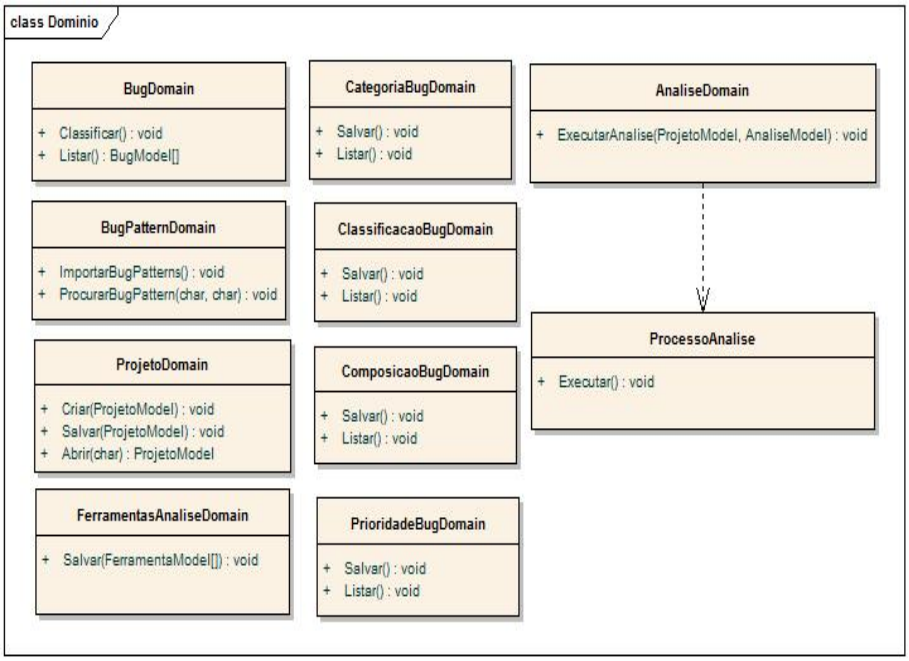

Figura 4.14: Diagrama de classes da camada de domínio

implementa o método `Execute` que executa um comando SQL qualquer e o método `GetDataReader`, que retorna um objeto chamado `DataReader`, utilizado em operações de leitura em banco de dados.

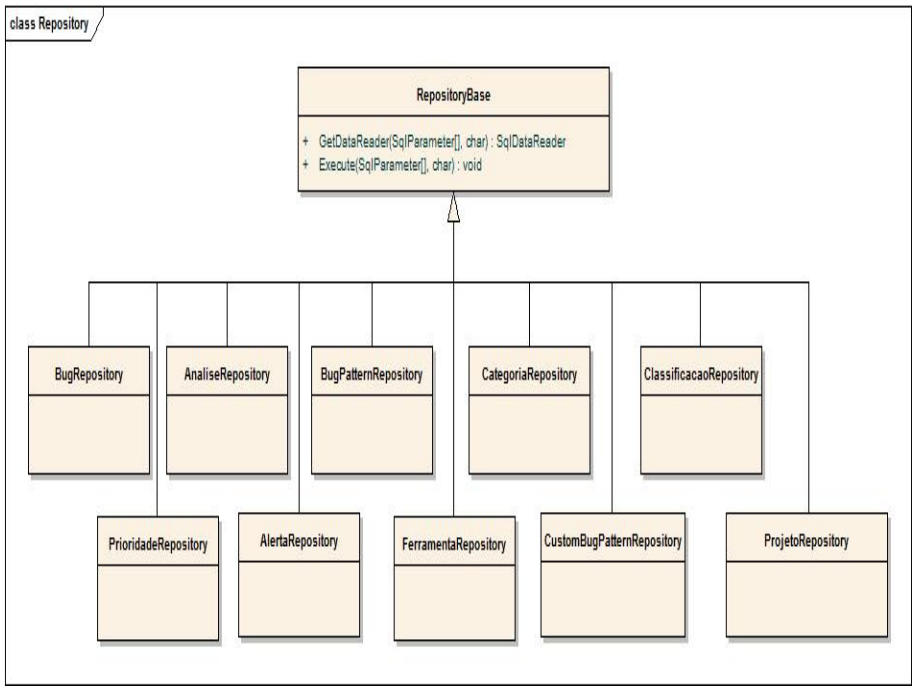

Figura 4.15: Diagrama de classes da camada de repositório

4. Camada de Modelo: Camada contendo as classes que mapeiam entidades do domínio da aplicação. Objetos de classes dessa camada, apresentadas no diagrama da Figura 4.16, trafegam desde a camada de apresentação até a camada de persistência. Como exemplo de classe dessa camada cita-se a classe `AlertaModel`, que representa um *bug* detectado por uma ferramenta para detecção de defeitos e ainda não consolidado pela ferramenta Smart Bug Detector. Após a fase de execução das ferramentas para detecção de defeitos, os alertas detectados serão processados a fim de eliminar duplicidades e aplicar as composições cadastradas. O resultado desse processamento é então tratado como *bug* e representado pela classe `BugModel`.

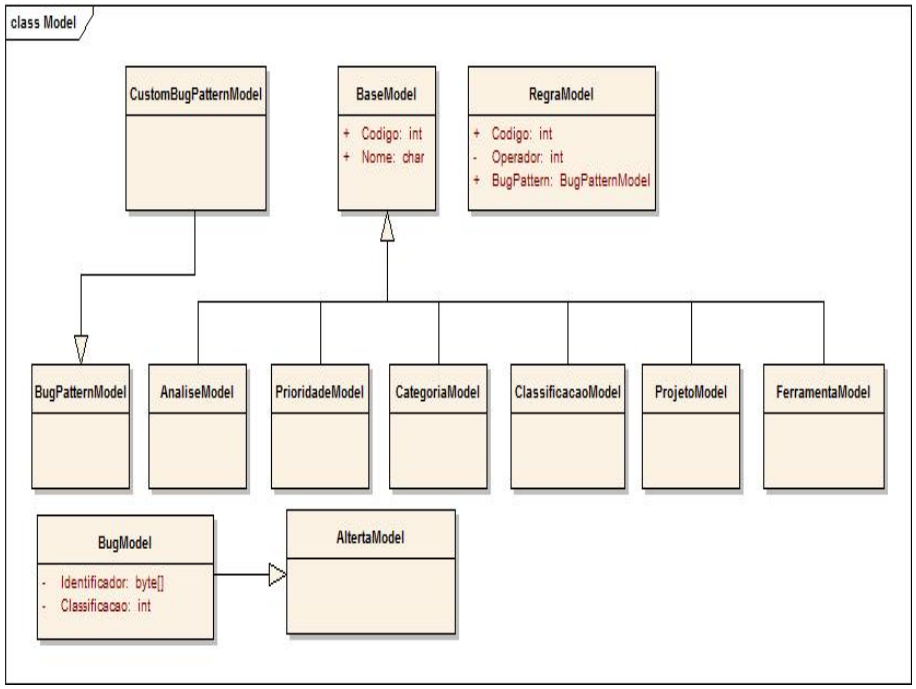

Figura 4.16: Diagrama de classes da camada de modelo

A Figura 4.17 apresenta o modelo físico do banco de dados utilizado na solução. De forma resumida apresenta-se a seguir o objetivo das principais tabelas contidas nesse modelo:

- *Bug*: Armazena os *bugs* consolidados pela ferramenta Smart Bug Detector. A maioria dos atributos dessa tabela foi definida com base na leitura do relatório de *bugs* das ferramentas para detecção de defeitos utilizadas na solução.

- Alerta: Essa tabela também recebe os *bugs* localizados pelas ferramentas para detecção de defeitos. No entanto, ela é utiliza de forma temporária apenas para auxiliar

no processo de consolidação de *bugs* que consistem em remoção de duplicidades e processamentos de composições. Ao término desse processamento, os dados dessa tabela são apagados.

- Bugpattern: Essa tabela contém os *bug patterns* extraídos das ferramentas para detecção de defeitos utilizadas na solução. As informações nela armazenadas são utilizadas pela funcionalidade de composição de *bugs*.

- Categoria: Contém as categorias de *bug patterns* extraídas das ferramentas para detecção de defeitos. Essa informação foi extraída visando sua utilização nas tarefas de ordenação de resultados e relatórios.

- Classificação: Contém a classificação de defeitos cadastrada na solução e utilizada na funcionalidade de classificação dos *bugs* detectados. A tabela *Bug* relaciona-se com a tabela Classificação, ou seja, sempre que um *bug* é classificado através da solução, o campo COD_CLASSIFICACAO na tabela *Bug* é atualizado.

- Projeto: Essa tabela contém os projetos de deteção de defeitos criados no Smart Bug Detector.

- Análise: Essa tabela contém as versões de detecção de defeitos criadas a partir de um projeto. Na solução atualmente implementada, cada execução realizada a partir de um projeto automaticamente cria-se uma nova versão.

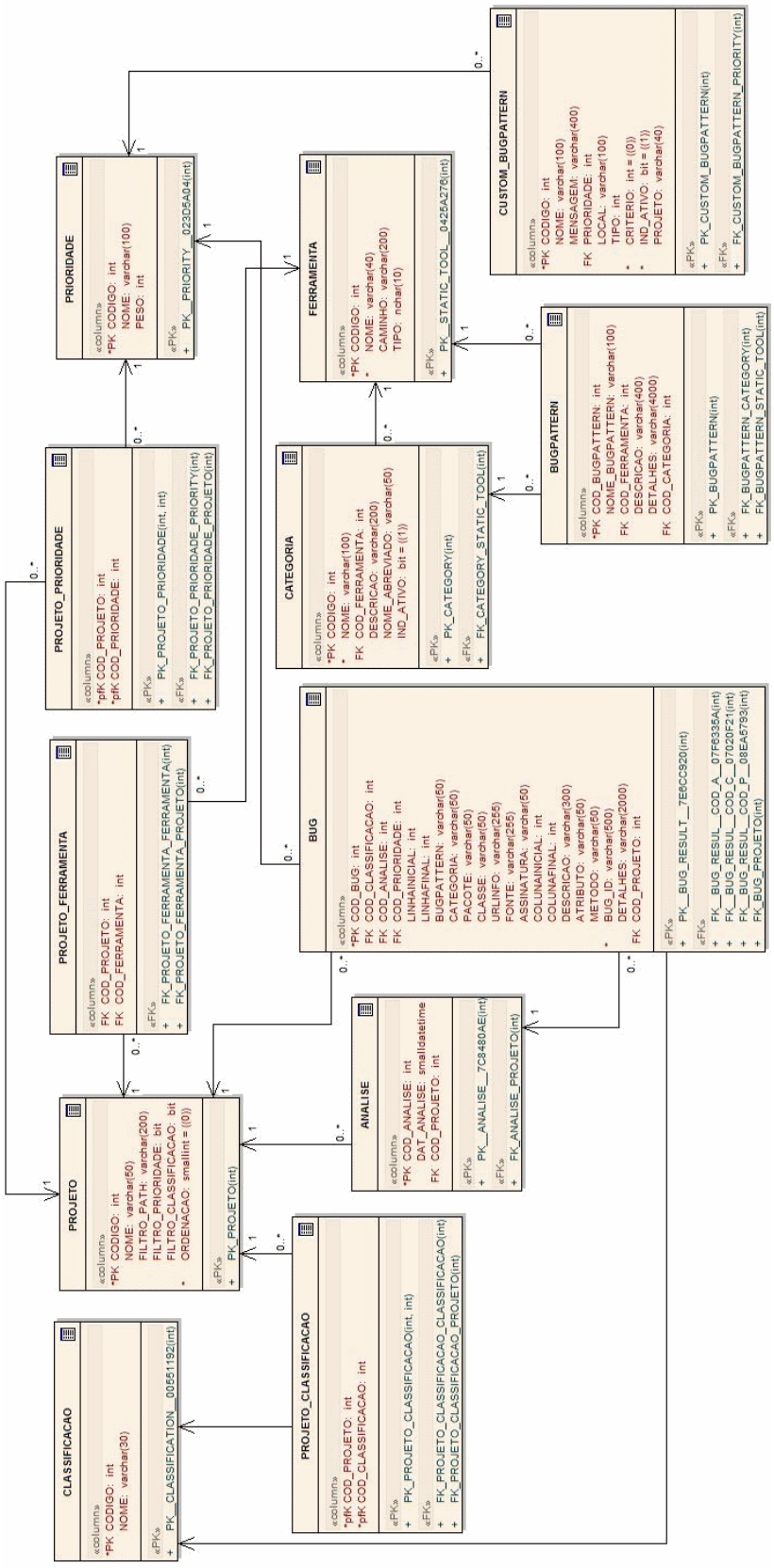

Figura 4.17: Diagrama ER da solução

De forma a exemplificar a interação entre as camadas da solução, apresenta-se na Figura 4.18 um diagrama de sequência com o fluxo de execução de um processo de detecção de defeitos. Primeiramente, o usuário comanda o início do processo através da tela principal na camada de apresentação. A classe `frmPrincipal` chama o método `ExecutarAnalise` da classe `AnaliseDomain`. O método `ExecutarAnalise`, por sua vez, invoca as ferramentas PMD e FindBugs, consolida os resultados removendo duplicidades e aplicando as composições e finalmente interage com a classe `BugRepository` para persistência dos resultados. A classe `BugRepository` após a persistência retorna como resultado um vetor de instâncias de `BugModel` para apresentação ao usuário.

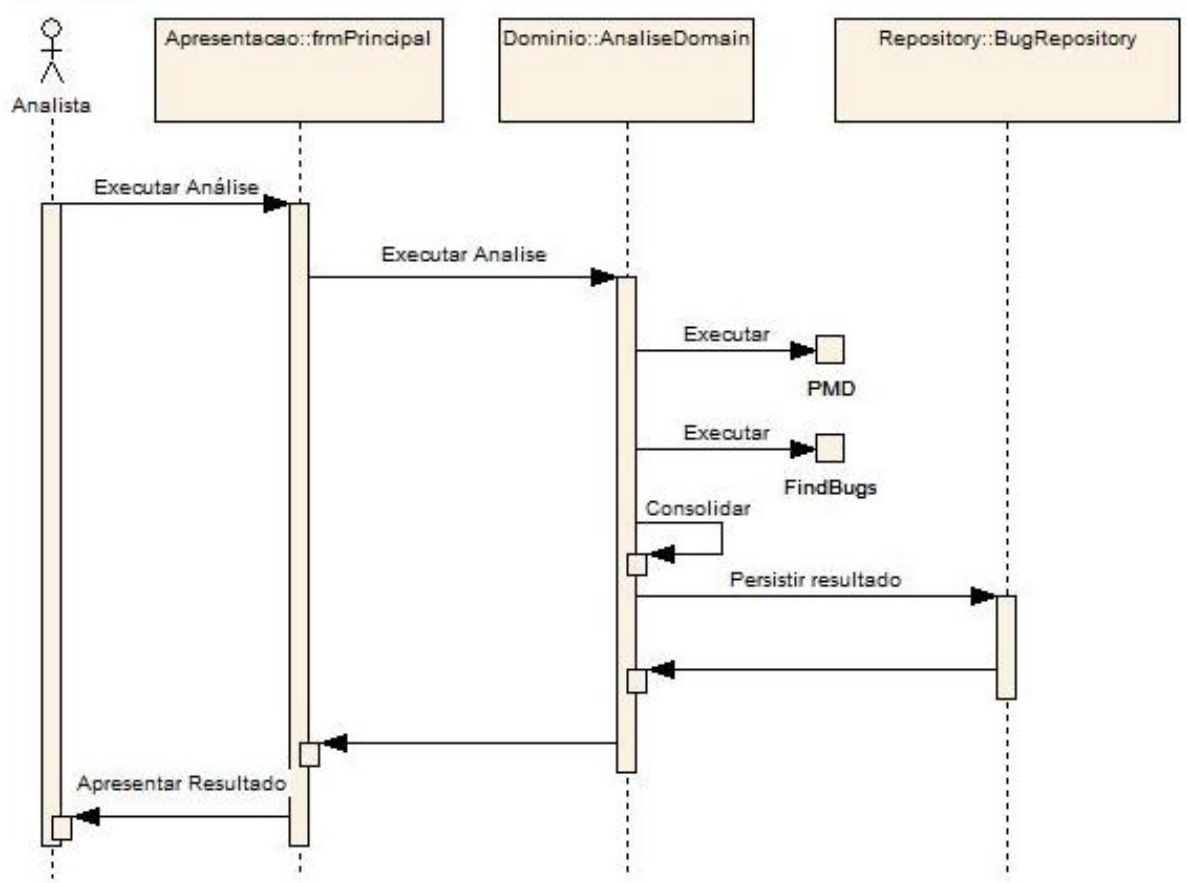

Figura 4.18: Diagrama de sequência do fluxo de detecção de defeitos

4.4 Estudo de Caso

Esta seção descreve um estudo de caso realizado com o objetivo de avaliar a ferramenta Smart Bug Detector. As seguintes atividades foram realizadas:

1. As ferramentas para detecção de defeitos FindBugs versão 1.3.8 e PMD versão 4.2.4 foram instaladas em um Notebook com processador Core 2 Duo de 1.66 GHz e 3 GB de memória RAM. Em seguida, essas ferramentas foram testadas individualmente de forma a certificar que estavam funcionando.

2. Na mesma máquina acima citada, intalou-se a ferramenta Smart Bug Detector e um banco de dados SQL Server. A ferramenta Smart Bug Detector foi então configurada para utilizar o FindBugs e o PMD. Após a configuração das ferramentas para detecção de defeitos, executou-se o processo de extração de categorias e *bug patterns*.

3. Selecionou-se o sistema Rhino, versão 1.43, como sistema alvo do processo de deteção de defeitos. O sistema Rhino foi selecionado devido a sua utilização em experimentos anteriores, descritos no Capítulo 3, onde se avaliou de forma individual o uso da ferramenta FindBugs.

4. Algumas categorias de *bug patterns* oferecidas pelo PMD foram desabilitadas na ferramenta Smart Bug Detector por serem de uso em situações específicas, como por exemplo, regras para migração de uma versão de JDK para outra, regras para aplicações J2EE e JavaBeans. Desta forma, a ferramenta Smart Bug Detector foi desenvolvida de modo que categorias de *bug patterns* desabilitadas não sejam consideradas ao se definir os parâmetros de execução das ferramentas para detecção de defeitos.

Os testes conduzidos no estudo de caso foram realizados em duas etapas, conforme descreve-se a seguir.

Execução do Smart Bug Detector sem cadastro de composições: Inicialmente a ferramenta Smart Bug Detector foi executada sem nenhuma composição cadastrada. Desta forma, a ferramenta detectou na versão 1.43 do Rhino um total de 637 *bugs*. O tempo de execução do processo de detecção de defeitos foi de 27 segundos. Os *bugs* detectados são apresentados na Tabela 4.1 onde observa-se que 90% dos *bugs* foram detectados por meio do PMD. Uma relação percentual semelhante foi observada no trabalho de comparação de ferramentas para detecção de defeitos de Rutar et al [RA04].

Execução do Smart Bug Detector após cadastro de composições: A fim de validar a utilização da ferramenta Smart Bug Detector após o cadastro de composições, definiu-se um conjunto de cinco composições, distribuídas conforme apresentado na Tabela 4.2.

Ferramenta	Categoria de Bug Pattern	Total de Bugs
FindBugs	BAD_PRACTICE	12
FindBugs	CORRECTNESS	6
FindBugs	MALICIOUS_CODE	7
FindBugs	MT_CORRECTNESS	4
FindBugs	PERFORMANCE	30
FindBugs	STYLE	6
Total		65
PMD	Basic Rules	59
PMD	Clone Implementation Rules	4
PMD	Coupling Rules	5
PMD	Design Rules	344
PMD	Finalizer Rules	1
PMD	Import Statement Rules	18
PMD	Security Code Guidelines	10
PMD	Strict Exception Rules	43
PMD	String and StringBuffer Rules	49
PMD	Type Resolution Rules	13
PMD	Unused Code Rules	26
Total		572
Total Geral		637

Tabela 4.1: *Bugs* detectados com o uso do Smart Bug Detector no sistema rhino 1.43

Essas composições foram criadas com base em análise dos *bugs* reportados na execução inicial. O critério utilizado para definição das composições foi:

- `Alias`: Selecionou-se um conjunto de *bug patterns* para verificação da funcionalidade de redefinição de prioridades e descrição de *bugs* detectados.

- `Or`: Para cada composição desse tipo, selecionou-se *bug patterns* de mesmo significado entre o PMD e o FindBugs como, por exemplo, *bug patterns* relacionados a declaração de atributos não utilizados.

- `And`: Para esse tipo, procurou-se agrupar em uma mesma composição *bug patterns* que, sozinhos não possuem sentido, porém agrupados dão origem a um *bug* relevante, como por exemplo, *bug patterns* voltados para detecção de falta de comentário de código. Dessa forma, os *bugs* reportados por essa categoria, podem ser vistos como um resumo de dois ou mais *bugs* detectados em uma classe ou método.

Após o cadastro das composições, executou-se a ferramenta Smart Bug Detector novamente sobre a versão 1.43 do Rhino. O tempo de execução observado foi de 28 segundos,

ou seja, o sistema demorou um segundo a mais para processar as composições e consolidar os resultados. Os *bugs* detectados por essas composições são apresentados na Tabela 4.3, onde observa-se um total de 598 *bugs* detectados sendo que 165 *bugs* foram reportados por meio das composições cadastradas e o restante, reportado diretamente pelo PMD e FindBugs. A diferença em relação ao total obtido sem o uso de composições ocorre em função do princípio de funcionamento das composições do tipo `Or` e `And`, conforme apresentado anteriormente. Por exemplo, observou-se que os dois *bugs patterns* utilizados na contrução da composição `Declaração de atributo não utilizado` reportaram inicialmente 31 `bugs`. Desses 31 *bugs*, seis deles detectados por *UUF_UNUSED_FIELD* estavam em duplicidade com outros seis *bugs* detectados por *UnusedPrivateField*. Assim, ao se criar uma composição `Or` com esses dois *bug patterns*, o número final de *bugs* foi reduzido de 12 para 6.

Tipo da Composição	**Nome da Composição**	**Bug Pattern**
Alias	Atribuição de valor em parâmetro	AvoidReassigningParameters
	Levantamento de exceção em bloco catch	PreserveStackTrace
And	Classe sem comentários	UncommentedEmptyMethod UncommentedEmptyConstructor
Or	Comparação de strings não recomendada	CompareObjectsWithEquals ES_COMPARING_PARAMETER_STRING_WITH_EQ
	Declaração de atributo não utilizado	UUF_UNUSED_FIELD UnusedPrivateField

Tabela 4.2: Exemplo de composições

Tipo da composição	**Bug Pattern**	**Bugs Detectados**
Atribuição de valor em parâmetro	Alias	78
Levantamento de exceção em bloco catch	Alias	29
Classe sem comentários	And	1
Comparação de strings não recomendada	Or	29
Declaração de atributo não utilizado	Or	25
Bugs detectados sem composição		439
Total		598

Tabela 4.3: *Bugs* detectados após o cadastro de composições

4.5 Considerações Finais

Com a utilização da ferramenta Smart Bug Detector observou-se que os objetivos inicialmente propostos foram alcançados. A meta-ferramenta, após o trabalho de configuração

inicial, permitiu utilizar de forma transparente as ferramentas FindBugs e PMD consolidando os resultados obtidos. O cadastro de composições apresentou benefícios em relação à flexibilização para criação de novas regras, redefinição de mensagens e prioridades.

Capítulo 5

Conclusões

5.1 Visão Geral da Solução Proposta

No Capítulo 3, apresentou-se um estudo de caso que teve dois objetivos principais:

- Avaliar o grau de efetividade de ferramentas para detecção de defeitos na localização de defeitos reportados por usuários finais.
- Avaliar se defeitos de alta prioridade detectados pelo FindBugs são removidos efetivamente em versões futuras de um sistema.

O resultado obtido na primeira parte do estudo foi negativo, ou seja, a ferramenta para detecção de defeitos não foi capaz de encontrar defeitos reportados por usuários finais. Na segunda parte do estudo, verificou-se que ferramentas para detecção de defeitos podem encontrar defeitos removidos em versões futuras. Desta forma, conclui-se que o uso dessas ferramentas durante o ciclo de desenvolvimento pode contribuir para a qualidade final do sistema.

No Capítulo 4, apresentou-se o projeto e a implementação da meta-ferramenta Smart Bug Detector. O principal objetivo dessa ferramenta é permitir a integração de duas ou mais ferramentas para detecção de defeitos. A solução foi proposta com base na verificação de que ferramentas para detecção de defeitos possuem um pequeno conjunto de detectores em comum e portanto o uso de duas ou mais ferramentas tende a aumentar a taxa de defeitos removidos. De fato, observa-se na literatura existente que empresas, como por exemplo Google e Nortel Networks, empregam mais de uma ferramenta para detecção de defeitos em seu ciclo de desenvolvimento de software [NWH+04, APM+07].

Desta forma, com a ferramenta proposta pretende-se oferecer uma solução que permita executar duas ou mais ferramentas de forma integrada e ao mesmo tempo proporcionar

funcionalidades como remoção de defeitos duplicados, criação de composições e consolidação de resultados. A versão do Smart Bug Detector descrita nessa dissertação foi implementada de forma a integrar o FindBugs e PMD. Como as ferramentas para detecção de defeitos diferem umas das outras em aspectos de arquitetura interna e forma de execução, a meta-ferramenta Smart Bug Detector não consegue reconhecer automaticamente uma nova ferramenta para detecção de defeitos, necessitando, portanto, de customização para isso.

5.2 Comparação com Trabalhos Relacionados

Avaliação de Ferramentas para Detecção de Defeitos: Ayewah et al. avaliaram os resultados gerados pelo FindBugs em três projetos de grande porte: Sun JDK, Sun Glassfish J2EE e parte do Google Java code base [APM+07]. Para cada projeto, eles classificaram manualmente os *bugs* da categoria corretude de prioridade média e alta da seguinte forma: trivial, com algum impacto funcional e com substancial impacto funcional. Por exemplo, dentre os 379 *bugs* obtidos ao avaliar o Sun JDK, somente 38 foram classificados como tendo substancial impacto funcional. No entanto, os autores deixam claro que a categorização utilizada é aberta a interpretações. Por exemplo, os gerentes de projeto do JDK podem ter um opinião diferente sobre a classificação proposta, baseados na sua experiência com o sistema, incluindo a importância dos módulos existentes e a experiência dos desenvolvedores de cada módulo.

Zheng et al. utilizaram o paradigma GQM (*Goals, Questions and Metrics*) para determinar quando a utilização de ferramentas para detecção de defeitos podem ajudar organizações a melhorar economicamente a qualidade de seus produtos de software [ZWN+06]. Para alcançar esse objetivo, eles avaliaram três projetos de grande porte em C++ da empresa Nortel Networks utilizando três ferramentas para detecção de defeitos comerciais: Gimpel's FlexeLint[1], Reasoning's Illuma[2] and Klockwork's inForce and GateKeeper[3]. Entretanto, não fica claro como o resultado obtido pode ser extrapolado para linguagens fortemente tipadas (como Java). Além disso, como a Nortel utiliza um serviço terceirizado para uma primeira análise dos *bugs* reportados, o trabalho utilizou somente os verdadeiros positivos que sobreviveram a esse serviço.

Wagner et al. avaliaram duas para detecção de defeitos – FindBugs e PMD – em dois projetos de uso industrial [WAWS08]. O principal objetivo foi determinar se essas

[1] `http://www.gimpel.com/html/flex.htm`.
[2] `http://www.reasoning.com`.
[3] `http://www.klockwork.com`.

ferramentas são efetivas em detectar defeitos de campo, como por exemplo, defeitos reportados por usuários finais desses sistemas. Os autores afirmam que não conseguiram relacionar nenhum *bug* reportado pelo PMD e FindBugs com *bugs* reportados por usuários finais. O motivo para isso é que defeitos de campo normalmente estão relacionados a regras de negócio do sistema. Apesar disso, eles afirmam que é importante avaliar *bugs* que apontam violações de práticas de programação recomendadas. Tais violações, podem não contribuir para um funcionamento incorreto do sistema, mas podem resultar em código difícil de manter e entender. Por isso, nessa dissertação, além de se avaliar a eficácia de ferramentas para detecção de defeitos em localizar defeitos reportados por usuários finais, avaliou-se também, a eficácia para localização de defeitos removidos em versões futuras.

Kim e Ernst propuseram um algoritimo de priorização de alertas reportados por ferramentas para detecção de defeitos [KE07]. Como motivação para o algoritimo proposto, ele avaliaram a precisão dos alertas reportados por três ferramentas para detecção de defeitos: FindBugs, PMD e JLint[4]). Essas ferramentas foram aplicadas sobre três programas de médio porte: Columba (um *cliente* de e-mail), Lucene (uma *engine* de pesquisa) e Scarab (um sistema de acompanhamento de ocorrências). Eles definiram o cálculo da precisão através da seguinte fórmula: (#alertas em linhas relacionadas ao *bug*) / (#alertas reportados pela ferramenta). A primeira parte da fórmula considera apenas aquelas linhas modificadas na correção de um *bug* ou outro problema reportado pelo usuário. Portanto, similar ao trabalho de Wagner et al. [WAWS08], a definição de precisão apresentada nesse trabalho não considera alertas removidos pelos desenvolvedores devido a otimizações internas ou adequação a boas práticas de programação.

Em um trabalho anterior, Kim e Ernst coletaram o tempo de vida de alertas em dois projetos de código livre (Columba e jEdit). Entretanto, eles deixam claro que seu experimento está sujeito a ambiguidades por considerarem o tempo de vida somente em nível de arquivos. Mais especificamente, eles computam o tempo de vida como o período entre a primeira e última ocorrência de uma categoria de *bug* em um arquivo. Se uma categoria possui múltiplos *bugs* em um arquivo, eles não consideram correções individuais. Além do mais, o resultado pode ser influenciado por refatorações como divisão de pacotes ou movimentação de classes e métodos de um arquivo para outro.

Integração de Ferramentas para Detecção de Defeitos: Na literatura pesquisada para elaboração desta dissertação, observou-se que empresas como Google e Nortel Networks utilizam mais de uma ferramenta para detecção de defeitos [APM+07, ZWN+06]. No entanto, esses trabalhos não mencionam a utilização de nenhuma técnica de integração.

[4] `http://jlint.sourceforge.net.`

Ayewah et al. citam, por exemplo, que no Google os *bugs* detectados pelo FindBugs e outras ferramentas são importados em banco de dados [APM+07]. No entanto, eles não citam a forma de execução das ferramentas e nem a forma de importação dos resultados para o banco de dados.

Wang et al. também descrevem uma solução para integração de ferramentas para detecção de defeitos denominada CODAS (*Code Defect Analysis Service*) [WMZ+08]. A ferramenta CODAS propõe a integração de ferramentas para detecção de defeitos por meio de um *web service*. O funcionamento da ferramenta CODAS pode ser resumido em três fases: *upload* do código, análise e retorno de resultados. Observa-se que a ferramenta proposta possui como limitação a necessidade de se realizar o *upload* de código para análise.

5.3 Contribuições

As principais contribuições desta dissertação são as seguintes:

1. Avaliação de ferramentas para detecção de defeitos. Esta dissertação avaliou o uso de ferramentas para detecção de defeitos para encontrar defeitos reportados por usuários finais. Acredita-se que os resultados e as informações disponibilizadas nesta dissertação possam ser utilizadas por mantenedores de software na tomada de decisão em relação ao uso de ferramentas para detecção de defeitos.

2. Projeto da meta-ferramenta Smart Bug Detector. A solução apresentada nesta dissertação permite a integração de ferramentas para detecção de defeitos ao mesmo tempo que proporciona funcionalidades como consolidação de resultados, redução de custo de configuração e execução simultânea de várias ferramentas. A solução proporciona também benefícios como aumento da taxa de detecção de defeitos reais, redefinição de prioridades e facilidade para compartilhamento de resultados obtidos.

5.4 Trabalhos Futuros

Como trabalho futuro, pretende-se prosseguir com pesquisas na área de qualidade de software. Assim, são descritas a seguir algumas das possíveis linhas de pesquisa:

- Realizar estudos de caso com novos sistemas, preferencialmente de diferentes domínios, como por exemplo, sistemas web.

- Aprimorar a implementação da Ferramenta Smart Bug Detector incorporando novas ferramentas para detecção de defeitos, como por exemplo, ferramentas para C/C++ e C#.

- Investigação do uso da Ferramenta Smart Bug Detector em outros sistemas, preferenciamente sistemas de médio e grande porte ainda em fase de desenvolvimento. Isso poderia fornecer dados sobre defeitos reais detectados possibilitando aprimorar a ferramenta para evidenciar esses tipos de defeitos.

- Desenvolvimento de um plug-in do Smart Bug Detector para IDE do Eclipse. Isso permitirá uma melhorar integração do processo de desenvolvimento com o processo de análise e correção de defeitos.

Bibliografia

[AHM+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 57–74, New York - NY, USA, 2006.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–8, New York - NY, USA, 2007.

[CBC+92] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael J. Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-processmeasurements. *IEEE Transactions on Software Engineerin*, 1:943–956, 1992.

[CGM06] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Jtl: the java tools language. *SIGPLAN Not.*, 41(10):89–108, 2006.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, 2005.

[DZ07] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *22th Conference on Automated Software Engineering (ASE)*, pages 433–436, New York - NY, USA, 2007.

[FHP07] Jeffrey S. Foster, Michael W. Hicks, and William Pugh. Improving software quality with static analysis. In *7th Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 83–84, New York - NY, USA, 2007.

[FRF+02] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[Joh77] S. C. Johnson. Lint: A C program checker. Technical Report 65, Bell Laboratories, dec 1977.

[KE07] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Foundations of Software Engineering (FSE)*, pages 45–54, New York - NY, USA, 2007.

[KNB08] R. Krishnan, Margaret Nadworny, and Nishil Bharill. Static analysis tools for security checking in code at motorola. *Ada Letters*, XXVIII(1):76–82, 2008.

[LBD+04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[Lou06] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[Mic09] Microsoft. FxCop home page. URL: http://msdn.microsoft.com/en-us/library/ bb429476(VS.80).aspx, 2009. Acessado em: 10/10/2009.

[NWH+04] Nachiappan Nagappan, Laurie Williams, John Hudepohl, Will Snipes, and Mladen Vouk. Preliminary results on using static analysis tools for software inspection. In *15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 429–439, Washington - DC, USA, 2004.

[PP94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transaction Software Engineering*, 20(6):463–475, 1994.

[PPV97] Dewayne E. Perr, Adam A. Porter, and Lawrence G. Votta. A primer on empirical studies (tutorial). In *Tutorial presented at 19th International Conference on Software Engineering (ICSE)*, pages 657–658, New York - NY, USA, 1997.

[QP09] QJ-Pro. Code analyzer for java. URL: http://qjpro.sourceforge.net, 2009. Acessado: 10/10/2009.

[RA04] Nick Rutar and Christian B. Almazan. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington - DC, USA, 2004. IEEE Computer Society.

[SHP06] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *2006 International Workshop on Mining Software Repositories*, pages 133–136, New York - NY, USA, 2006.

[WAWS08] Stefan Wagner, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *2008 International Conference on Software Testing, Verification, and Validation*, pages 248–257, Washington - DC, USA, 2008.

[WJK+05] Stefan Wagner, Jan Jürjens, Claudia Koller, Peter Trischberger, and Technische Universität München. Comparing bug finding tools with reviews and tests. In *17th International Conference on Testing of Communicating Systems*, pages 40–55. Springer, 2005.

[WMZ+08] Qianxiang Wang, Na Meng, Zhiyi Zhou, Jinhui Li, and Hong Mei. Towards soa-based code defect analysis. In *2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 269–274, Washington - DC, USA, 2008.

[ZWN+06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions On Software Engineering*, 32(4):240–253, 2006.