Uma Meta-Ferramenta para Detecção de Defeitos

Sílvio José de Souza[1], **João Eduardo Montandon**[1], **Marco Túlio Valente**[2]

[1]Instituto de Informática, PUC Minas
[2]Departamento de Ciência da Computação, UFMG

silviojsouza@gmail.com, joao.montandon@sga.pucminas.br, mtov@dcc.ufmg.br

Resumo. *O uso combinado de ferramentas para detecção de defeitos pode ampliar a taxa de defeitos que refletem verdadeiras falhas. Porém, essa forma de uso somente é viável com uma solução que permita integrar essas ferramentas fornecendo funcionalidades que solucionem ou minimizem os problemas gerados pela combinação de suas características. Assim, descreve-se neste artigo o projeto e a implementação da ferramenta Smart Bug Detector, uma meta-ferramenta que permite o uso integrado de duas ferramentas para detecção de defeitos, fornecendo funcionalidades que minimizam os problemas gerados a partir da sua integração. Além disso, apresenta-se um estudo de caso desenvolvido com o objetivo de avaliar os benefícios proporcionados pela meta-ferramenta.*

1 Introdução

Recentemente, tem crescido o interesse tanto acadêmico como industrial pelo emprego de técnicas e ferramentas para detecção de defeitos [8, 5, 1, 3]. Essas ferramentas – também chamadas de *bug findings tools* – funcionam procurando por violações de padrões de programação. Como exemplo dos defeitos detectados por essas ferramentas pode-se citar o acesso a referências *null*, uso inapropriado de métodos (como *equals*, *clone* etc), uso incorreto de primitivas de sincronização, *overflow* em vetores, dentre outros. Em resumo, tais ferramentas ampliam e sofisticam a capacidade de detecção de defeitos normalmente obtida por técnicas como inspeção e testes.

Mesmo com o crescente interesse por técnicas automatizadas de detecção de defeitos, observa-se que as ferramentas existentes possuem um pequeno conjunto de detectores em comum. Ou seja, o uso combinado de duas ou mais ferramentas para detecção de defeitos pode aumentar a taxa de defeitos que denotam verdadeiras falhas [11, 10]. De fato, observa-se na literatura que algumas empresas, como Google e Nortel Networks, empregam mais de uma ferramenta para detecção de defeitos em seu ciclo de desenvolvimento de software [9, 2].

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*.

Desta forma, descreve-se neste artigo 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 ferramentas para detecção de defeitos, fornecendo funcionalidades que minimizam problemas como: a dificuldade para o uso de duas ou mais

ferramentas para detecção de defeitos [11, 10]; o grande número de defeitos reportados [2]; e a eficiência do critério de priorização de defeitos [6].

O restante deste trabalho está estruturado conforme descrito a seguir. A Seção 2 descreve a ferramenta Smart Bug Detector, incluindo suas principais características e funcionalidades. Já a Seção 3 relata um estudo de caso realizado com a ferramenta. A Seção 4 descreve trabalhos relacionados, e por fim, a Seção 5 apresenta as conclusões deste trabalho.

2 Smart Bug Detector

Esta seção apresenta uma ferramenta, chamada Smart Bug Detector, que funciona como uma interface entre o usuário e diversas ferramentas para detecção de defeitos. Mais especificamente, a meta-ferramenta proposta se destina 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 [11]. No entanto, o seu uso combinado é complexo devido ao esforço necessário para configurar, executar e analisar os resultados obtidos pelas diversas ferramentas. Assim, a solução apresentada tem como objetivo principal permitir o uso de duas ou mais ferramentas, detectando defeitos duplicados e consolidando os resultados gerados em um formato único.

- **Grande número de defeitos reportados:** Ferramentas para deteção de defeitos geram um grande número de defeitos que normalmente são armazenados em arquivos XML ou exibidos 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 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 [6]. Como solução, a meta-ferramenta Smart Bug Detector permite alterar as prioridades dos defeitos reportados pelas ferramentas utilizadas.

Considerando os problemas descritos acima, as principais funcionalidades da ferramenta Smart Bug Detector são as seguintes:

- **Execução de uma ou mais ferramentas para detecção de defeitos:** Permite executar uma ou mais ferramentas para detecção de defeitos sem que o usuário tenha que lidar com os detalhes de uso inerentes a cada ferramenta.

- **Composição de *bug pattern*:** Principal funcionalidade da ferramenta, permite criar novos *bug patterns* por meio da composição com os *bug patterns* extraídos das ferramentas para detecção de defeitos. *Bug pattern* é o termo normalmente usado para designar os tipos e padrões de defeitos detectados por ferramentas de análise estática [5, 1].

- **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.

- **Flexibilidade para definição de prioridades e classes de defeitos:** Permite o cadastro de novas prioridades e classes de defeitos.

- **Classificação de defeitos de forma compartilhada:** Possibilita o uso compartilhado facilitando a divisão do trabalho de análise e classificação de defeitos reportados.

- **Exibição de métricas:** Disponibiliza métricas sobre detecções de defeitos realizadas.

A Figura 1 apresenta a visão principal da interface do sistema, a qual é dividida em três áreas principais. O painel superior à esquerda exibe uma árvore com os defeitos detectados em uma ordem definida pelo usuário. O painel superior à direita apresenta o código fonte, quando disponível, responsável pelo defeito. Já o painel inferior exibe detalhes sobre o defeito selecionado.

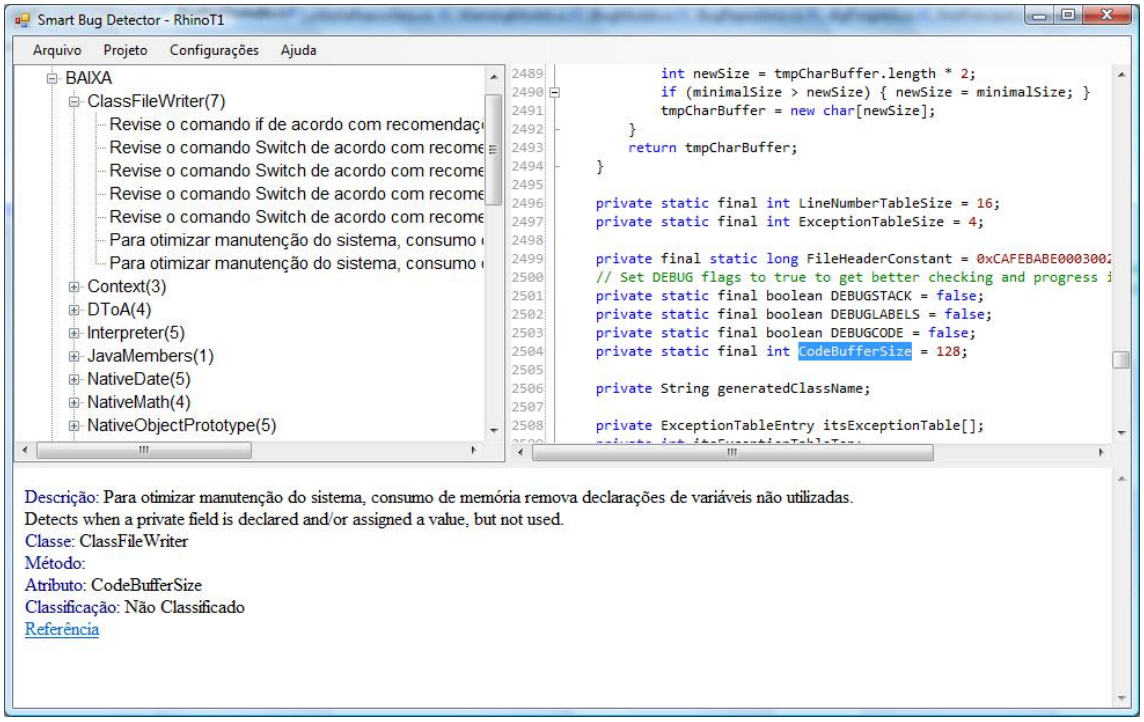

Figura 1. Interface da ferramenta Smart Bug Detector

2.1 Composição de *Bug Patterns*

Principal funcionalidade da ferramenta Smart Bug Detector, a composição de *bug patterns* permite que novas regras de detecção de *bugs* sejam criadas a partir dos detectores oferecidos pelas ferramentas para detecção de defeitos. As composições de *bug patterns* que podem ser criadas são as seguintes:

- `Alias`: Permite redefinir um *bug pattern* implementado por uma ferramenta para detecção de defeitos, atribuindo-lhe uma nova descrição ou prioridade. A Figura 2 exemplifica o princípio de funcionamento desse tipo de composição. Nessa figura, observa-se que para todo *bug* $d1$ detectado pelo *bug pattern* da composição `Alias`, um *bug* R é reportado. Ou seja, a composição `Alias` permite o mapeamento de um *bug* $d1$ para um bug R definido pelo usuário. Essa funcionalidade é útil para a tradução de mensagens ou para torná-las mais intuitivas para o usuário.

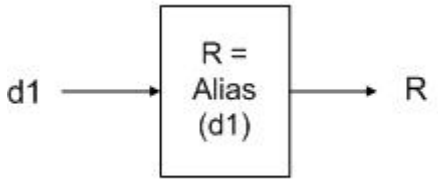

Figura 2. Composição `Alias`

- `Or`: Permite combinar *bug patterns* das ferramentas para detecção de defeitos utilizando o operador `or`. Esse tipo de composição permite remoção de duplicidade de *bugs* detectados ao se combinar os resultados de diferentes ferramentas para detecção de defeitos. A Figura 3 apresenta o princípio de funcionamento dessa composição.

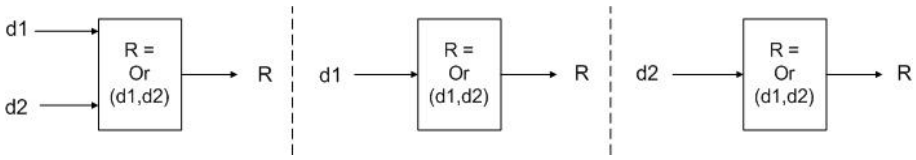

Figura 3. Composição `Or`

- `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. O princípio de funcionamento desse tipo de composição é ilustrado na Figura 4.

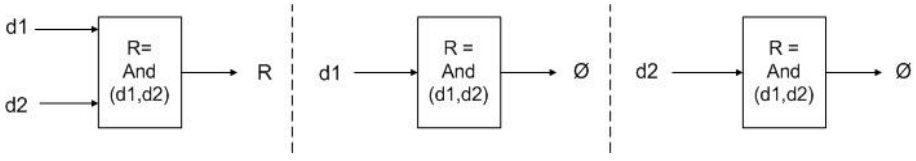

Figura 4. Composição `And`

No momento da definição de uma composição, é possível especificar um escopo onde essa composição será válida. 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.

3 Estudo de Caso

Com o objetivo de avaliar a ferramenta e o potencial de suas composições, foi realizado um estudo de caso no qual a ferramenta Smart Bug Detector foi configurada para utilizar as ferramentas FindBugs (versão 1.3.8) e PMD (versão 4.2.4). O sistema Rhino (versão 1.43), um interpretador de Javascript, foi escolhido como sistema alvo do processo de detecção dos defeitos. Durante a configuração das ferramentas, alguns detectores de *bug patterns* do PMD foram desabilitados por terem aplicação apenas em situações muito específicas, como por exemplo, padrões aplicáveis à migração de versões de JDK. O estudo de caso foi conduzido em duas etapas, descritas a seguir.

Execução 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 foi de 27 segundos. Os *bugs* detectados são apresentados na Tabela 1. Pode-se observar que 90% dos *bugs* foram detectados por meio do PMD.

Ferramenta	**Categoria de Bug Pattern**	**Total**
FindBugs	BAD_PRACTICE	12
FindBugs	CORRECTNESS	6
FindBugs	MALICIOUS_CODE	7
FindBugs	MT_CORRECTNESS	4
FindBugs	PERFORMANCE	30
FindBugs	STYLE	6
FindBugs	**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
PMD	**Total**	**572**
Total Geral		**637**

Tabela 1. *Bugs* **detectados com o uso do Smart Bug Detector no sistema Rhino 1.43**

Execução após o cadastramento de composições: A fim de avaliar a utilização da ferramenta Smart Bug Detector após o cadastro de composições, definiu-se um conjunto de cinco composições, conforme apresentado na Tabela 2. 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`: 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`: Agrupou-se *bug patterns* que sozinhos não possuem sentido, porém agrupados dão origem a um *bug* mais relevante. Por exemplo, *bug patterns* voltados para detecção de falta de comentário de código.

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. Os *bugs* detectados por essas composições são apresentados na Tabela 3. Nessa tabela, observa-se um total de 598 *bugs* detectados, sendo que 165 *bugs* foram reportados por meio das composições cadastradas. A diferença em relação ao resultado anterior ocorre devido ao funcionamento das composições do tipo `Or` e `And`. Por exemplo, observou-se que os dois *bugs patterns* utilizados na composição `Declaração de atributo não utilizado` reportaram inicialmente 31 *bugs*. Dentre esses *bugs*, seis bugs do tipo *UUF_UNUSED_FIELD* – reportados pelo FindBugs – estavam em duplicidade com outros seis *bugs* do tipo *UnusedPrivateField* – reportados pelo PMD . 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 2. Exemplos de composições

Bug Pattern	**Tipo da composição**	**Total**
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 3. *Bugs* **detectados após o cadastro de composições**

4 Trabalhos Relacionados

Rutar et al. examinaram os resultados reportados por cinco ferramentas de detecção de defeitos sobre vários sistemas Java [10]. A experiência conduzida mostrou uma baixa correlação entre os defeitos levantados pelas ferramentas (31% entre FindBugs e PMD). Além disso, os autores tiveram dificuldade em gerir a grande quantidade de avisos reportados pelas mesmas. Por isso, eles propõem como trabalho futuro a implementação de uma meta-ferramenta que examine um sistema utilizando várias ferramentas de detecção, possibilitando um manuseio simplificado dos avisos gerados e permitindo a combinação de *bugs* de várias ferramentas. Wagner et al. realizam um estudo de caso analisando os *bugs* reportados a partir da análise de dois sistemas internos de uma empresa de consultoria [11]. Utilizando as ferramentas FindBugs e PMD, o trabalho procura responder questões acerca do uso das ferramentas durante o processo de desenvolvimento de *software*. Dentre as muitas conclusões relatadas no trabalho, os autores destacam que, pelo fato das ferramentas possuírem poucos detectores em comum, o seu uso combinado pode contribuir para aumentar a precisão dos *bugs* reportados. Os trabalhos de Rutar et al. e de Wagner et al. serviram de inspiração para desenvolvimento da ferramenta Smart Bug Detector.

Existem atualmente algumas outras ferramentas similares ao Smart Bug Detector, como, por exemplo, a ferramenta Sonar [4]. Essa ferramenta oferece uma plataforma de gerenciamento de qualidade que permite a integração de programas diversos para a análise de qualidade (como FindBugs, PMD e CheckStyle). No entanto, a ferramenta Sonar não suporta o conceito de composição de *bug patterns*, implementado no Smart Bug Detector. RepoGuard é um *framework* para integração entre ambientes de desenvolvimento e sistemas de controle de versões [7]. O objetivo é permitir que um processo extra seja executado sempre que ocorrer uma operação em um sistema de controle de versões. Esse processo extra pode incluir, por exemplo, a execução de uma (ou de mais de uma) ferramenta de detecção de defeitos. No entanto, RepoGuard não oferece recursos para combinar os resultados caso múltiplas ferramentas de detecção de defeitos sejam executadas.

5 Conclusões

Neste trabalho, descreveu-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.

Além disso, a composição de *bug patterns* – funcionalidade presente no Smart Bug Detector – mostrou-se um recurso que pode ser explorado em diversos aspectos. Como, por exemplo, na validação e criação de novos defeitos, baseados nos *bug patterns* existentes.

A versão do Smart Bug Detector descrita neste trabalho 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 novas

ferramentas para detecção de defeitos, necessitando de customização para isso. Como trabalho futuro, pretende-se investigar soluções que permitam atenuar esse esforço de customização. Pretende-se também disponibilizar uma versão da meta-ferramenta que funcione com um *plug in* do ambiente de desenvolvimento Eclipse.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq. Gostaríamos de agradecer a César Couto pela revisão de uma versão preliminar do artigo.

Referências

[1] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5), 2008.

[2] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *7th Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–8, 2007.

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications ACM*, 53(2):66–75, 2010.

[4] Simon Brandhof, Olivier Gaudin, Freddy Mallet, and Evgeny Mandrikov. Sonar home page. http://www.sonarsource.org, Visitado em 13/04/2010.

[5] 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, 2007.

[6] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first? In *15th International Symposium on Foundations of Software Engineering (FSE)*, pages 45–54, 2007.

[7] Malte Legenhausen, Stefan Pielicke, Jens Ruhmkorf, Heinrich Wendel, and Andreas Schreiber. Repoguard: A framework for integration of development tools with source code repositories. In *4th IEEE International Conference on Global Software Engineering (ICGSE)*, pages 328–331, 2009.

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

[9] 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, 2004.

[10] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256, 2004.

[11] Stefan Wagner, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *1st International Conference on Software Testing, Verification, and Validation (ICST)*, pages 248–257, 2008.