Ferramentas para Análise Estática de Códigos Java

Ricardo Terra
Instituto de Informática
PUC Minas
Anel Rodoviário Km 23,5 – 31.980-110
Belo Horizonte, Brazil
rterrabh@gmail.com

Roberto S. Bigonha
Departamento de Ciência da Computação
UFMG
Campus da Pampulha – 31.270-010
Belo Horizonte, Brazil
bigonha@dcc.ufmg.br

ABSTRACT
Nowadays, a large portion of the systems which have come into production presents errors. To minimise this problem, Verification and Validation (V&V) becomes indispensable. Many techniques have been developed to automatically find errors in systems. Static code analysis is one of the instruments known by software engineering for mitigation of errors, either to perform style verification or bugs checking or both. *Checkstyle*, *FindBugs*, *PMD*, *Klocwork Developer* are some of the various automatic tools for the Java language which help to reduce the number of errors using static code analysis. This paper makes a comparison of these tools, evaluating them and measuring the effectiveness and efficiency by the application of several case studies.

RESUMO
Atualmente, uma grande parcela de sistemas que entram em produção apresenta erros. Para minimizar este problema, as atividades de Verificação e Validação (V&V) se tornam indispensáveis. Várias técnicas têm sido desenvolvidas para automaticamente encontrar erros nos sistemas. Análise estática de código é um dos instrumentos conhecidos pela engenharia de software para a mitigação de erros, seja por sua utilização para a verificação de regras de estilos, para a verificação de erros ou ambos. *Checkstyle*, *FindBugs*, *PMD*, *Klocwork Developer* são algumas das diversas ferramentas automáticas para a linguagem Java que ajudam a reduzir o número de erros utilizando análise estática de código. Este artigo realiza a comparação dessas ferramentas, avaliando-as e mensurando a eficácia e eficiência a partir da aplicação de diversos estudos de caso.

Palavras-Chaves
Engenharia de software, validação, verificação, inspeção, testes, erros, regras de estilo, análise estática de código, Java.

1. INTRODUÇÃO
Definitivamente, a construção de software não é uma tarefa simples. Pelo contrário, pode se tornar bastante complexa, dependendo das características e dimensões do sistema a ser criado. Por isso, está sujeita a diversos tipos de problemas que acabam na obtenção de um produto diferente daquele que se esperava [9]. Vários fatores podem ser identificados como causas destes problemas, mas a maioria deles tem origem no erro humano [11, 9]. A construção de software depende principalmente da habilidade, da interpretação e da execução das pessoas que o constroem e, por isto, erros acabam surgindo, mesmo com a utilização de métodos e ferramentas de engenharia de software.

Para que estes erros possam ser encontrados antes de o software entrar em produção, existem uma série de atividades, em conjunto conhecidas como "Verificação e Validação" ou "V&V", que possuem a finalidade de garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o especificado [10, 14].

O restante deste artigo está organizado conforme descrito a seguir. Seção 2 contextualiza o artigo ilustrando as técnicas de V&V. Seção 3 aborda a análise estática de código. Seção 4 apresenta as ferramentas de análise estática de código. Seção 5 apresenta uma avaliação das ferramentas. Por fim, na Seção 6 são expostas as considerações finais.

2. VERIFICAÇÃO E VALIDAÇÃO (V&V)
A princípio, verificação e validação podem parecer a mesma coisa, porém existe uma sucinta diferença entre elas. Verificação se destina a mostrar que o projeto do software atende à sua especificação, enquanto que validação se destina a mostrar que o software realiza exatamente o que o usuário espera que ele faça [2].

O objetivo principal do processo de V&V é estabelecer confiança de que o sistema de software está adequado ao seu propósito. O nível de confiabilidade exigido depende de fatores como *função do software*, *expectativas do usuário*, *ambiente* e *mercado*.

Segundo Sommerville [14], existem duas abordagens complementares para a verificação e validação do sistema:

- *Inspeções de software* analisam e verificam representações de sistema como documentos de requisitos, diagramas de projeto e código fonte do programa. Revisões de código, análises automatizadas e verificação formal são técnicas de V&V estáticas[1].
- *Testes de software* envolvem executar uma implementação do software com dados de teste para examinar as saídas e o comportamento operacional. O teste é

[1]Técnicas estáticas são aquelas que não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas [9, 5, 14].

uma técnica de V&V dinâmica[2].

Revisão de código consiste em examinar o programa, em buscas de erros, omissões ou anomalias, sem executá-lo. Geralmente são dirigidas por *checklists* de erros e heurísticas que identificam erros comuns em diferentes linguagens de programação. Para alguns erros e heurísticas, é possível automatizar o processo de verificação de programas em relação a essa lista, o que possibilitou as análises automatizadas. Por outro lado, verificação formal consiste em validar ou invalidar a correção dos algoritmos subjacentes a um sistema com relação a uma certa propriedade ou especificação formal, utilizando métodos formais [14].

As técnicas de inspeções de software podem somente verificar a correspondência entre um programa e sua especificação, isto indica que elas não podem demonstrar que o software é útil operacionalmente, nem utilizá-las para verificar propriedades emergentes como desempenho e confiabilidade.

Os testes de software por sua vez possuem dois tipos: o *teste de defeitos* que tem a finalidade de encontrar inconsistências entre um programa e sua especificação, isto é, revelar defeitos e o *teste de validação* que tem a finalidade de mostrar que o software é o que o cliente deseja [14]. Contudo, o teste pode mostrar a presença de erros, mas não demonstrar que não existam erros remanescentes [3].

No entanto, inspeções e testes não são abordagens concorrentes, mas sim, complementares. Cada uma tem vantagens e desvantagens que devem ser utilizadas em conjunto no processo de V&V. Por exemplo, é comum um projeto de desenvolvimento de software iniciar seu processo de V&V com inspeções e, uma vez que um sistema esteja integrado, testá-lo para verificar as propriedades emergentes e se a funcionalidade do sistema é a que seu proprietário realmente deseja.

3. ANÁLISE ESTÁTICA DE CÓDIGO (AEC)

Popularmente, o termo Análise Estática de Código se refere à análise automatizada, que é uma das técnicas estáticas de inspeção de software no processo de V&V. Logo, o termo verificador ou analisador estático de código é usualmente aplicado à verificação realizada por uma ferramenta automatizada seguida de uma análise humana sobre os resultados.

Os verificadores estáticos são ferramentas de software que varrem o código fonte ou mesmo o código objeto (no caso da linguagem Java, o *bytecode*). Cada uma das versões tem suas próprias vantagens. Quando você trabalha diretamente sobre o código fonte do programa, suas verificações refletem exatamente o código escrito pelo programador. Compiladores otimizam o código e o *bytecode* resultante pode não refletir o código fonte. Por outro lado, trabalhar em *bytecode* é consideravelmente mais rápido, o que é importante em projetos contendo dezenas de milhares de linhas de código [8].

A intenção dos verificadores estáticos é chamar a atenção para anomalias do programa, como variáveis usadas sem serem iniciadas, variáveis não usadas ou atribuições cujos valores poderiam ficar fora de seus limites. As anomalias são, freqüentemente, um resultado de erros de programação ou omissões, de modo que elas enfatizam coisas que poderiam sair erradas quando o programa fosse executado. Contudo, essas anomalias não são, necessariamente, defeitos de programa, o que justifica a necessidade da análise humana.

Pode-se enumerar quatro benefícios da utilização de verificadores estáticos [6]:

1. Encontra erros e códigos de risco;
2. Fornece um retorno objetivo aos programadores para ajudá-los a reconhecer onde eles foram precisos e onde eles foram desatentos;
3. Fornece a um líder de projeto uma oportunidade para estudar o código, o projeto e a equipe sob uma perspectiva diferente;
4. Retira certas classes de defeitos, o que possibilita que a equipe concentre-se mais nas deficiências do projeto.

3.1 Linhas de Atuação

Os verificadores estáticos de código, que são ferramentas automáticas para a verificação de código, podem verificar regras de estilos de programação, erros ou ambos. Um verificador de regras de estilo examina o código para determinar se ele contém violações de regras de estilo de código. Por outro lado, um verificador de erro utiliza a análise estática para encontrar código que viola uma propriedade de correção específica e que pode causar um comportamento anormal do programa em tempo de execução [5].

Um verificador de regras de estilo (*style checker*, em inglês) verifica a conformação de um código fonte à sua definição de estilo de programação. Ao contrário de um programa *Beautifier*[3], um verificador de regras de estilo realiza apenas uma validação do conteúdo, não alterando a formatação original do documento. Porém, regras de estilo, tais como "sempre colocar constantes no lado esquerdo de comparações" e "não utilizar instruções de atribuição na condição de instruções *if*", podem ajudar a prevenir certos tipos de erros e a melhorar a qualidade do software. Entretanto, violações dessas orientações de estilo não são particularmente susceptíveis de serem erros.

Ao contrário de um verificador de regras de estilo, um verificador de erro (*bug checker*, em inglês) é uma ferramenta voltada para a detecção automática de erros que é desenvolvida tomando como base as tendências comuns dos desenvolvedores em atrair defeitos que, em sua maioria, não são visíveis aos compiladores [8]. Estes defeitos se justificam pelo fato de todos poderem cometer erros tolos e das linguagens de programação oferecerem oportunidades para erros latentes. Contudo, os resultados de tal ferramenta nem sempre são

[2]Técnicas dinâmicas são aquelas que se baseiam na execução de um programa ou de um modelo [9, 5, 14].

[3]Um programa *Beautifier*, traduzido por Formatador de Código Fonte, diz respeito a um sistema que, a partir do código fonte de um programa de computador, gera o código formatado seguindo certas regras, sem ocorrer alteração da semântica do programa.

```
1  public class LeituraArquivo
2  {
3    public void ler(String nomeArquivo){
4      String linha;
5      try {
6        BufferedReader in =
7          new BufferedReader(
8            new FileReader(nomeArquivo));
9
10       while (in.ready()){
11         if ((linha = in.readLine()) != ":q")
12           conteudo += linha;
13       }
14     } catch (IOException e) {
15     }
16   }
17
18   String conteudo;
19 }
```

Figura 1: Exemplo de uma classe Java de leitura de arquivo com alguns defeitos.

defeitos reais, mas podem ser vistos como um alerta de que um pedaço de código é crítico de alguma forma [16].

A Figura 1 exibe uma classe Java que realiza a leitura de um arquivo que foi submetida a um verificador de regras de estilo e a um verificador de erro com o intuito de deixar mais claro a diferença entre eles. O verificador de regras de estilo alertou sobre problemas como a abertura de chaves de um bloco (linha 2 e 11), falta de comentário *Javadoc* (linha 1 e 3), variáveis com nome reduzido (linha 6), atribuições em subexpressões (linha 11), bloco *catch* vazio (linha 14-15), visibilidade não explicitada (linha 18) e inexistência de ordem na declaração dos membros da classe. Por outro lado, o verificador de erro alertou sobre problemas como a comparação de *strings* usando '!=' (linha 11), concatenação de *strings* em laços de repetição (linha 12) e fluxo de leitura não encerrado.

Logo, não existem impedimentos de se utilizar, em um mesmo projeto, uma ferramenta de verificação de erro em conjunto com uma ferramenta de verificação de regras de estilo, pois elas se complementam. Tanto é possível, que existem ferramentas de integração de verificadores estáticos de código [12].

3.2 Comparativo com outras técnicas de V&V

Diversas pesquisas têm sido realizadas na procura de defeitos em código utilizando análise estática automatizada, porém poucos estudos têm sido realizados sobre o comparativo destes verificadores de erro com outras técnicas estabilizadas de detecção de defeitos, tais como revisão de código ou teste de software.

3.2.1 Revisão de Código

A revisão de código é uma das técnicas de inspeções de software, assim como a análise estática automatizada. Wagner [16] constatou que a revisão de código encontrou todos os tipos de defeitos encontrados por um verificador de erro, porém houve uma disparidade no número de erros encontrados em alguns tipos de defeitos. O motivo dessa disparidade pode ser facilmente explicado tomando como exemplo dois tipos de erros bem simples: "variáveis inicializadas, mas não utilizadas" e "claúsula *if* não necessária". Um erro do tipo "variáveis inicializadas, mas não utilizadas" é um problema facilmente computável o que indica que um verificador de erro é bem mais eficaz, bastando possuir uma rotina que verifica se todas as variáveis inicializadas foram utilizadas. Em contrapartida, um erro do tipo "claúsula *if* não necessária" é um problema resolvido a partir do entendimento da lógica de um programa o que o torna mais propício de ser encontrado por um processo de revisão de código.

Em síntese, a revisão de código mostra-se mais bem-sucedida do que as ferramentas de verificação de erro, pois ela é capaz de detectar mais tipos de defeitos. Contudo, é recomendável primeiramente utilizar uma ferramenta de verificação de erro antes de revisar o código, o que faz com que os defeitos que são encontrados por ambos já estejam removidos. Isso acontece porque a automação os faz de forma mais barata e mais minuciosa do que uma revisão manual.

3.2.2 Teste de software

O teste de software é uma técnica dinâmica, o que já faz com que se espere que os erros encontrados sejam diferentes daqueles encontrados por técnicas estáticas. Wagner [16] realizou um comparativo entre sistemas de software que apresentaram defeitos e constatou que não houve um mesmo defeito que tenha sido encontrado tanto com testes quanto com ferramentas de verificação de erro. Os testes revelaram vários defeitos relacionados à utilização das funcionalidades do sistema. Por outro lado, as ferramentas revelaram defeitos que os testes não foram capazes de encontrar, pois são defeitos que somente podem ser encontrados por testes extensos de estresse como, por exemplo, uma conexão ao banco de dados não encerrada.

Em síntese, os testes de software e as ferramentas de verificação de erro detectam defeitos diferentes. Testes de software são eficazes em encontrar defeitos lógicos que são mais bem visualizados quando o software está em execução, enquanto que as ferramentas de análise estática automatizada são eficazes em encontrar defeitos relacionados aos princípios de programação e à manutenibilidade. Portanto, é altamente recomendável a utilização de ambas as técnicas.

4. FERRAMENTAS

A idéia de ferramentas de análise estática não é nova. Em 1970, Stephen Johnson da *Bell Laboratories*, escreveu *Lint*, uma ferramenta para examinar código fonte de programas C que tenham compilado sem erros e verificar erros que tenham escapado ao compilador. Os compiladores não realizavam verificação sofisticada de tipo, especialmente entre programas compilados separadamente. *Lint* forçava as regras de tipos e várias restrições de portabilidade mais restritamente que os compiladores [7].

O sucesso de *Lint* deu aos programadores de hoje várias ferramentas descendentes, tanto de código aberto quanto proprietárias, que englobam diferentes linguagens. Contudo, a abordagem dessas ferramentas em linguagens modernas é bem diferente, já que as próprias linguagens apresentam tipagem forte, portabilidade e não utilizam várias características propensas a erros, tais como variáveis não inicializadas, uso de instruções *goto*, instruções de gerenciamento de memória, etc.

4.1 Características

Existem importantes atributos que uma ferramenta de análise estática deve ter [1, 4, 5]:

- *Consistência*[4]: Todo *erro real* é reportado pela análise, isto é, nenhum erro real deixará de ser reportado.
- *Integridade*[5]: Todo erro reportado é um erro real, isto é, nenhum erro reportado é um *falso positivo*.
- *Utilidade*[6]: Se reporta erros que são relevantes.

Os atributos consistência e integridade não são requisitos para uma ferramenta de análise estática, simplesmente pelo fato de as técnicas de V&V concorrentes, tal como revisões de código e testes de software, não serem nem consistentes nem íntegras. Ainda assim, esses atributos são cada vez mais visados, tanto que a percentagem de falso positivo para cada erro real é um indicador importante da efetividade de uma ferramenta. Certamente, uma ferramenta consistente, íntegra e útil que realizasse uma verificação funcional completa do programa seria a mais eficaz ferramenta de análise estática automatizada, porém existem dificuldades claras de computação, esforço e custo para que isto se realizasse [4].

4.2 Técnicas

Conforme já abordado na Seção 3, a análise estática em aplicações Java pode ser aplicada sobre o código fonte e/ou *bytecode* da aplicação. As ferramentas que atuam sobre o *bytecode* geralmente utilizam a Biblioteca de Engenharia de *Bytecode* (BCEL) do projeto *Apache Jakarta*. Cada uma destas versões tem suas vantagens, porém atuar sobre as duas versões tirando proveito das vantagens particulares de cada uma apresenta-se como a melhor solução.

Embora cada verificador de código trabalhe de seu próprio modo, a maioria deles compartilham algumas características básicas. Eles lêem o programa e constroem algum modelo dele, um tipo de representação abstrata (*Abstract Syntax Tree*, por exemplo) que eles poderão utilizar para detectar os padrões de erro que eles reconhecem. Esses verificadores também desenvolvem algum tipo de análise de fluxo de dados, tentando inferir os valores possíveis que as variáveis possam ter em certos pontos do programa [13, 5, 8].

4.3 Principais ferramentas

A Tabela 1 exibe um breve sumário das principais ferramentas para código Java que foram alvo de estudos nos últimos anos. As ferramentas destacadas em negrito foram avaliadas em nosso estudo. A ferramenta *ESC/Java2* foi descartada da avaliação, pois necessita de anotações em código que faz com que tenha características diferentes. Ainda foram descartadas as ferramentas *Jlint* e *QJ-Pro*, pois tiveram seu desenvolvimento interrompido.

Checkstyle é uma ferramenta voltada à verificação de regras de estilo, sendo ideal para projetos que querem forçar os desenvolvedores a seguirem um estilo de programação. É altamente configurável permitindo customizar as regras de estilo por projeto. Atualmente, ela também provê algumas funcionalidades de um verificador de erro, tais como verificações de problemas de projeto de classes, código duplicado e padrões de erros.

Nome	*Versão*	*Data de liberação*	*Licença*
Checkstyle	**4.4.1**	**dez/2007**	**Livre**
ESC/Java2	2.0.4	jan/2008	Livre
FindBugs	**1.3.4**	**maio/2008**	**Livre**
Jlint	3.1	out/2006	Livre
Klocwork Developer	**7.6.0.7**	**fev/2007**	**Proprietária**
PMD	**4.2.2**	**maio/2008**	**Livre**
QJ-Pro	2.2.0	mar/2005	Livre

Tabela 1: Principais ferramentas para análise estática de código Java

FindBugs é uma popular ferramenta voltada à verificação de erros desenvolvida pela *University of Maryland*. Atualmente, ela provê suporte a mais de 250 padrões de erros sendo mais de uma centena deles classificados como erros de correção. Além disso, permite que programadores escrevam seus próprios padrões de erros.

Klocword Developer é uma ferramenta voltada à verificação de erros que abrange a tecnologia de análise estática. Além de encontrar defeitos, ela também procura por vulnerabilidades de segurança e ainda executa métricas e análises arquiteturais.

PMD é uma ferramenta voltada tanto à verificação de regras de estilo quanto à verificação de erros. Iniciou-se como um simples verificador de regras de estilo até serem adicionadas várias funcionalidades de um verificador de erro. Atualmente, permite que programadores escrevam seus próprios padrões de erros, do mesmo modo como ocorre no *FindBugs*.

Com exceção do *Klocwork Developer*, todas as demais ferramentas são de código aberto. O *Checkstyle* e o *PMD* trabalham sobre o código fonte, *FindBugs* sobre o *bytecode* e o *Klocwork Developer* sobre ambos o que o torna mais abrangente. É importante mencionar que todas as ferramentas avaliadas possuem módulo de extensão (*plug-in*, em inglês) para a versão mais atual da IDE Eclipse conforme pode ser observado na Figura 2.

5. AVALIACÃO

Neste artigo, foram selecionadas, como estudos de caso, algumas regras de estilo largamente utilizadas e alguns erros frequentemente cometidos em projetos de software para avaliar a eficácia das ferramentas de análise estática [15]. As Tabelas 2 e 3 apresentam uma breve descrição do defeito e os resultados da aplicação de cada ferramenta em um código que continha esse defeito. O símbolo '$\surd$' indica que a ferramenta alertou sobre o defeito e o símbolo '-' indica que a ferramenta não alertou sobre o defeito.

Em relação às regras de estilo, a ferramenta *Checkstyle* apresentou o melhor resultado, seguida da ferramenta *PMD*. As ferramentas *FindBugs* e *Klocwork Developer* praticamente

[4]Consistência: *soundness*, em inglês.
[5]Integridade: *completeness*, em inglês.
[6]Utilidade: *usefulness*, em inglês.

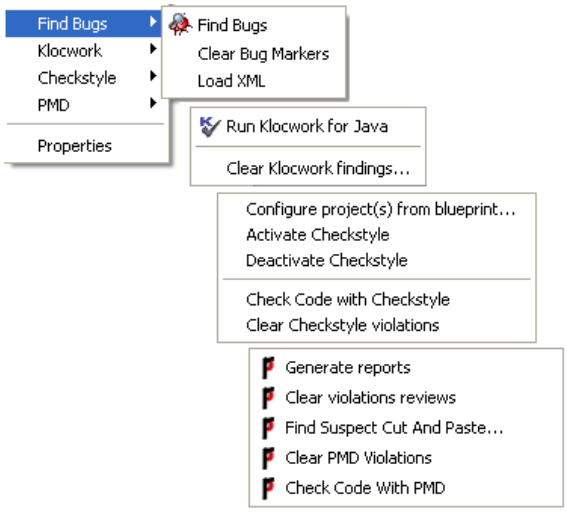

Figura 2: Módulos de extensão para IDE Eclipse

Regra de estilo	CS	FB	KW	PMD
Bloco *catch* vazio	✓	-	✓	✓
Atribuição em subexpressões	-	-	-	✓
Comparação com constante	-	-	-	-
Instrução vazia	✓	-	-	✓
Ordem de declaração	✓	-	-	-
Padrão de nomenclatura	✓	-	-	-
Posição da cláusula *default* em instruções *switch*	✓	-	-	✓
Uso da referência *this*	✓	-	-	-
Uso de chaves	✓	-	-	✓
Total	7/9	0/9	1/9	5/9

Tabela 2: Resultado da aplicação das ferramentas em regras de estilo largamente utilizadas

não alertaram sobre as regras de estilo, o que era esperado, já que ambas possuem o propósito de verificação de erros, não de regras de estilo. Em relação aos erros, a ferramenta *Klocwork Developer* apresentou um resultado ideal, tendo encontrado erros em todos os estudos de caso. Logo em seguida, encontra-se a ferramenta *FindBugs* que somente não foi capaz de encontrar erro em um dos estudos de caso. As ferramentas *PMD* e *Checkstyle* apresentaram resultados insatisfatórios.

Até mesmo no *Lint*, havia ocasiões em que o programador realmente desejava adotar um padrão de implementação pouco convencional, o que resultava em um falso positivo. Em apenas um estudo de caso, um único falso positivo foi alertado por uma ferramenta, o que é uma excelente taxa de integridade. Contudo, estudos de caso mais complexos são necessários para efetivamente mensurar a integridade dessas ferramentas.

Se os alertas referem-se a erros reais, mas são defeitos com que os desenvolvedores não se preocupam ou não querem se preocupar, basta a ferramenta prover um filtro de quais problemas devem ser verificados, o que foi encontrado em todas as ferramentas avaliadas neste estudo. Por outro lado,

Erro	CS	FB	KW	PMD
Conexão com banco de dados não encerrada	-	✓	✓	✓
Valor de retorno ignorado	-	✓	✓	-
Perda de referência nula (*null dereference*, em inglês)	-	✓	✓	✓
Concatenação de *strings* em laços de repetição	-	-	✓	✓
Fluxo não encerrado	-	✓	✓	-
Não utilização de operador booleano de curto-circuito	-	✓	✓	-
Objetos iguais têm o mesmo código de dispersão (*hashcode*, em inglês)	✓	✓	✓	✓
Uso de '==' ao invés de '*equals*'	✓	✓	✓	✓
Comparação com nulo redundante	-	✓	✓	-
Total	2/9	8/9	9/9	5/9

Tabela 3: Resultado da aplicação das ferramentas em erros frequentemente cometidos

se os alertas referem-se a falso positivos, o problema não está mais na configuração da ferramenta, mas sim na própria ferramenta. Uma possível solução seria o uso de anotações, entretanto a sua inserção requer um esforço adicional por parte dos programadores [4, 12].

Com exceção de um único estudo de caso, todos os outros foram alertados por pelo menos uma das ferramentas avaliadas. O fato de existir um problema que não foi alertado por nenhuma ferramenta, reforça o fato da ausência de alertas de uma ferramenta não implicar na ausência de erros do programa. Isso nos remete ao conceito de que nenhum verificador estático de código é consistente e íntegro, ou seja, nenhum verificador pode nos assegurar que um programa está correto – tal garantia não é possível [8].

6. CONSIDERAÇÕES FINAIS

A análise de código-fonte automatizada é uma das técnicas de inspeções de software. Ela surgiu da possibilidade de automatizar o processo de verificação em relação a certos itens do *checklist* de uma revisão de código manual. Portanto, a técnica de revisão de código encontra todos os tipos de defeitos encontrados por um verificador estático de código, porém existe uma disparidade no número de erros encontrados em alguns tipos de defeitos. Por outro lado, os testes de software detectam defeitos totalmente diferentes, que são mais relacionados à utilização das funcionalidades do sistema.

Assim, as ferramentas de análise estática automatizada são eficazes em encontrar defeitos relacionados aos princípios de programação estruturada e à manutenibilidade, o que a torna uma excelente prática de programação defensiva [17]. Isso ocorre porque a programação defensiva envolve tanto encontrar defeitos no código pela identificação de inconsistências quanto aperfeiçoar o código para aumentar a legibilidade e a manutenibilidade. Esses princípios evitam o código defensivo em excesso, o qual tornaria o produto menos legível, mais lento e bem mais complexo.

Logo, uma possível metodologia para a qualidade de um projeto de software seria utilizar uma ferramenta de verificação

estática sempre após a compilação e antes de uma revisão de código manual para remover os defeitos que são encontrados por ambas as técnicas, uma vez que a ferramenta realiza essa verificação de forma mais barata e mais minuciosa. Desse modo, o programa obtém uma indicação inicial de correção adequada para a realização de testes.

Em relação às ferramentas, a ferramenta *Checkstyle* apresentou-se como a melhor ferramenta de verificação de regras de estilo. Por outro lado, a ferramenta *Klocwork Developer* foi, sem dúvida, a melhor ferramenta de verificação de erros alertando sobre todos os erros, contudo possui a desvantagem de ser uma ferramenta proprietária. Isso faz com que a ferramenta *FindBugs*, que é gratuita e comprovadamente quase tão eficaz, seja uma excelente opção.

As ferramentas verificadoras de regras de estilo tendem a, cada vez mais, incorporar verificadores para outros propósitos. Um exemplo disso são as ferramentas *Checkstyle* e *PMD* que tinham o foco de verificação de regras de estilo, porém incorporaram tantas outras funcionalidades que pode-se dizer que são ferramentas mistas. Isso, por um lado, pode ser bom por serem cada vez mais abrangentes, porém cuidado deve ser tomado para evitar que acabe por não fazer nenhuma das funções corretamente, como foi o caso do *PMD* que apresentou um resultado pouco notável tanto na verificação de regras de estilo quanto na verificação de erros. Por outro lado, as ferramentas verificadoras de erros mantêm seu foco e não estão incorporando verificadores para outros propósitos.

Enfim, nenhuma máquina pode substituir o bom senso, o sólido conhecimento dos fundamentos, o pensamento claro e a disciplina, entretanto ferramentas de verificação estática de código podem efetivamente ajudar os desenvolvedores e, a sua mais ampla adoção, pode aumentar significativamente a qualidade do software [5].

7. AGRADECIMENTOS

Gostaríamos de agradecer ao Marco Túlio Valente a revisão deste artigo, aos acadêmicos Alexander Flávio de Oliveira e Lucas Martins do Amaral as suas colaborações e ao Danny Baldwin da equipe *Klocwork* a prestatividade e as informações fornecidas.

8. REFERÊNCIAS

[1] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Vancouver, 2002. ACM.

[2] B. W. Boehm. Software engineering; r & d trends and defense needs. In *Research directions in software technology*, pages 1–9, Cambridge, 1979. MIT Press.

[3] E. W. Dijkstra, O. J. Dahl, other, et al. *Structured Programming*. Academic Press, Londres, 1972.

[4] C. Flanagam, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlim, jun. 2002. ACM.

[5] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, dez. 2004.

[6] R. Jelliffe. Mini-review of java bug finders. *O'Reilly Developer Weblogs*, mar. 2004.

[7] S. C. Johnson. Lint, a c program checker. *Bell Laboratories*, pages 1–11, dez. 1977.

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

[9] J. C. Maldonado, M. E. Delamaro, and M. Jino. *Introdução ao Teste de Software*, chapter 1, pages 1–7. Elsevier, Rio de Janeiro, 2007.

[10] N. Parekh. Software verification & validation model - an introduction, 2005.

[11] R. S. Pressman. *Engenharia de Software*. McGraw-Hill, São Paulo, 5 edition, 2002.

[12] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, 2004. IEEE Computer Society.

[13] SAP AG, Alemanha. *Open Source Static Analysis Tools for Security Testing of Java Web Applications*, dez. 2006.

[14] I. Sommerville. *Engenharia de Software*. Pearson Addison-Wesley, São Paulo, 8 edition, 2007. caps. 22–23, p. 341–373.

[15] R. Terra. Ferramentas para análise estática de códigos java. Monografia, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, fev. 2008. Disponível em: http://www.ricardoterra.com.br/publicacoes/ufmg-faecj.pdf.

[16] S. Wagner, J. Jurjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th International Conference on Testing of Communicating Systems (TestCom'05)*, volume 3502, pages 40–55, Springer, 2005. ACM.

[17] M. Zaidman. Teaching defensive programming in java. *J. Comput. Small Coll.*, 19(3):33–43, 2004.