FACULDADE DE MINAS – FAMINAS-BH

CURSO DE SISTEMAS DE INFORMAÇÃO

JOSÉ OSWALDO DOS SANTOS NETO
RODRIGO CAMPOS SANTANA

ANÁLISE ESTÁTICA DE CÓDIGO JAVA: UM COMPARATIVO ENTRE *FindBugs* e *Klocwork Developer*

BELO HORIZONTE

2008

JOSÉ OSWALDO DOS SANTOS NETO
RODRIGO CAMPOS SANTANA

ANÁLISE ESTÁTICA DE CÓDIGO JAVA: UM COMPARATIVO ENTRE *FindBugs* e *Klocwork Developer*

Trabalho de conclusão de curso apresentado ao Curso de Sistemas de Informação da Faculdade de Minas – FAMINAS-BH, como requisito de avaliação para conclusão do curso.

Orientador: Prof. Ricardo Terra

BELO HORIZONTE

2008

José Oswaldo dos Santos Neto
Rodrigo Campos Santana

Análise estática de código Java:
Um comparativo entre *Findbugs* e *Klocwork Developer*

Objetivo: Identificar entre as ferramentas *FindBugs* e *Klocwork Developer* qual a melhor opção para realizar Análise Estática de Código

FAMINAS-BH – Faculdade de Minas
Curso de Sistemas de Informação

Área de concentração: Engenharia de software, análise estática de código.

Data de aprovação: _____ / _____/ _____

Prof. Ricardo Terra
Orientador

_____

Prof. Paulo Henrique Fernandes de Matos
Coordenador do Curso de Sistemas de Informação

_____

Dedicamos este trabalho aos nossos pais, que sempre nos apoiaram e encorajaram para superarmos as dificuldades que enfrentamos.

AGRADECIMENTOS

Gostariamos de agradecer a todos professores, pelas inúmeras ajudas que nos deram durante todo o curso de graduação.

Em especial, aos Professores Ricardo Terra e Maria Carolina, pela enorme contribuição e orientação no desenvolvimento deste trabalho.

Ao nosso amigo Cleber Barbosa, pela fundamental contribuição no empréstimo de algumas obras citadas neste trabalho.

Aos nossos familiares, amigos e colegas de trabalho, pelo apoio dado em todos os momentos que precisamos.

Em especial, aos nossos pais, por tudo que fizeram e fazem por nós e por serem um exemplo a seguir.

E, por último, à Deus, que nos ilumina, guia e nos da força em todos os dias de nossa vida.

“O primeiro passo em direção ao sucesso é o conhecimento.”

Nicola Tesla

RESUMO

No contexto de desenvolvimento de um programa, a atividade de teste é um elemento importante na construção de um software. Ao codificar programas, desenvolvedores cometem alguns erros que não são identificados por compiladores, portanto é fundamental a aplicação de técnicas de Verificação e Validação (V&V). Uma das técnicas de inspeção de software do processo de V&V é a Análise Estática de Código (AEC) que se refere à análise realizada de forma automática por uma ferramenta que varre o código fonte de um programa, auxiliando o compilador na detecção de anomalias. Existem várias ferramentas que realizam AEC na linguagem Java, destacando-se *Checkstyle*, *FindBugs*, *Klocwork Developer* e *PMD*. Assim, este trabalho tem como principal objetivo identificar, dentre as ferramentas *FindBugs* e *Klocwork Developer*, qual é a melhor opção para quem pretende realizar AEC em Java, levando em consideração os resultados da aplicação dessas ferramentas em alguns erros usualmente cometidos por desenvolvedores.

Palavras chave: Engenharia de Software, Teste de Software, Análise Estática de Código, Java, Erro, *FindBugs*, *Klocwork Developer*

ABSTRACT

To develop an application, the test activity is an important element in the software construction. When coding programs, developers make mistakes which are not identified by compilers, therefore is essential the implementation of Verification and Validation (V & V). One of the software inspection techniques of the V & V is the Static Code Analysis (SCA) which refers to the analysis carried out by an automated tool that scans the source code of an application, helping the compiler to detect anomalies. There are several tools that perform SCA in the Java language, especially Checkstyle, FindBugs, Klocwork Developer, and PMD. Thus, this work aims to identify, between the FindBugs and Klocwork Developer, the best option for those who want to do SCA in Java, taking into account the results of applying these tools in some errors usually performing by developers.

Keywords: Software Engineering, Software Test, Static Code Analysis, Java, Error, *FindBugs*, *Klocwork Developer*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

A E C Análise Estática de Código

I D E *Integrated Development Environment*, traduzido por Ambiente Integrado de Desenvolvimento

JVM *Java Virtual Machine,* traduzido por Maquina Virtual Java

V & V Verificação e Validação

XML *Extensible Markup Language*, traduzido por Linguagem de Marcação Extensível

SUMÁRIO

1 INTRODUÇÃO

Pressman (2005, p. 786) afirma que, no processo de construção de um software, a atividade de teste de software é um elemento crítico da garantia de qualidade de software.

Esses testes devem começar o quanto antes e para que isso aconteça são aplicadas técnicas de Verificação e Validação (V & V), mais precisamente inspeções no código fonte.

> A verificação refere-se ao conjunto de atividades que garante que o software implemente corretamente uma função específica, já a validação refere-se a um conjunto diferente de atividades que garante que o software que foi construído é rastreável às exigências do cliente. (PRESSMAN, 2005, p. 836)

Segundo Sommervile (2007, p. 346), inspeções no código fonte têm o objetivo de detectar defeitos de programa, essa idéia vem desde a década de 1970, formalizada pela primeira vez na IBM. Hoje é um método de verificação amplamente usado.

> De forma resumida, o processo tradicional de inspeção envolve o planejamento da inspeção, indivíduos revisando um determinado artefato, um encontro em equipe para discutir e registrar os defeitos, a passagem dos defeitos para o autor do artefato para que possam ser corrigidos e uma avaliação final sobre a necessidade de uma nova inspeção. (KALINOWSKI, 2008)

Ao codificar um programa, o programador pode cometer erros não identificados pelo compilador. Portanto, faz-se necessário inspecionar o código fonte para identificar tais erros que os analisadores estáticos de código são aptos a buscar.

Defeitos geralmente ocultam outros erros, logo, é fundamental realizar uma inspeção prévia no código eliminando essas anomalias e evidenciando possíveis erros que poderiam deixar de serem capturados.

Ainda segundo Sommerville (2007, p. 345), existem vários estudos e experimentos que têm demonstrado que as inspeções são mais eficientes para descobrir defeitos do que teste de programas.

Ao realizar análise estática de codigo, mais precisamente em Java, pretende-se que o resultado seja consistente e íntegro, isto é, reportará todos os erros

existentes e, todos os erros reportados, são realmente erros. Por isso, é necessário utilizar a melhor ferramenta a fim de atingir tal objetivo. Assim, a questão a ser respondida neste trabalho é: qual a melhor ferramenta, dentre *FindBugs* e *Klocwork Developer*, para realizar análise estática de código em Java?

Neste trabalho, uma pesquisa relacionada a teste de software, mais precisamente análise estática de código (AEC) em Java, será realizada. Ele identificará dentre as ferramentas *FindBugs* e *Klocwork Developer,* qual é a melhor opção para quem pretende realizar AEC em Java, levando em consideração o resultado apresentado no comparativo que será realizado.

Especificamente pretende-se selecionar os principais erros que são usualmente cometidos por desenvolvedores de sistemas em Java, para que posteriormente possamos realizar uma análise comparativa desses erros utilizando essas ferramentas.

Atualmente, é evidente a valorização da qualidade de software. Pádua (2003, p. 7) afirma, que geralmente a qualidade de um produto decorre diretamente da qualidade do processo utilizado na produção dele. Possuir uma boa ferramenta que auxilie nesse processo é fundamental. Portanto, ao se comparar tais ferramentas, esta pesquisa se justifica por contribuir para a base de conhecimento científico da área.

Dessa forma, no Capítulo 2 deste trabalho são abordadas a Análise Estática de Código, algumas das principais ferramentas existentes e de forma mais detalhada as ferramentas *FindBugs* e *Klocwork Developer.* No Capítulo 3 é realizado um estudo de caso que tem como finalidade mensurar a eficácia e a eficiência das ferramentas analisadas. Isto será realizado aplicando as ferramentas em erros previamente selecionados e, em seguida, será realizado um comparativo dos resultados obtidos. E, por fim, no Capítulo 4 são apresentadas as considerações finais.

2 ANÁLISE ESTÁTICA DE CÓDIGOS (AEC)

Terra e Bigonha (2008, p. 3), definem o termo Análise Estática de Código (AEC) como referente a análise automatizada, que é uma técnica de V&V. Conseqüentemente, o termo verificador ou analisador estático de código é aplicado à verificação realizada por uma ferramenta automatizada.

Sommerville (2007, p. 345), define analisadores estáticos como ferramentas de software que varrem o texto-fonte de um programa e detectam possíveis defeitos e anomalias. Atuam de forma a complementar os recursos de detecção de erros providos pelo compilador da linguagem. Ele ainda os classifica como:

- **Defeitos de dados.** São exemplos de defeitos de dados: as variáveis usadas antes da inicialização, as variáveis declaradas, mas nunca usadas, as variáveis atribuídas duas vezes, mas nunca utilizadas entre atribuições, possíveis violações de limites de vetor, as variáveis não declaradas.
- **Defeitos de controle.** Pode-se entender como defeitos de controle: os códigos inacessíveis, as ramificações incondicionais em *loops,* entre outros.
- **Defeitos de entrada e saída.** Exemplificados pelas variáveis geradas duas vezes sem tarefa de impedimento.
- **Defeitos de interface.** Os defeitos de interface são os tipos de parâmetros que não combinam, os números de parâmetros que não combinam, os resultados de funções não usadas e as funções e procedimentos não chamados.
- **Defeitos de gerenciamento de armazenamento** (não se aplica, pois em Java não existe ponteiros). Exemplos desse tipo de defeitos são: ponteiros não atribuídos e operações aritméticas de ponteiros.

2.1 Ferramentas de Análise Estática de Código

Várias ferramentas realizam AEC em Java, Terra e Bigonha (2008, p. 4) apresentam uma lista com as principais existentes:

Nome	Versão	Data Liberação	Licença
Klocwork Developer	7.6.0.7	fev/2007	Proprietária
FindBugs	1.3.5	setembro/2008	Livre
Jlint	3.1	out/2006	Livre
QJ-Pro	2.2.0	mar/2005	Livre
Checkstyle	5.0 beta1	jul/2008	Livre
PMD	4.2.3	agosto/2008	Livre

Tabela 1: Principais ferramentas de AEC.
Fonte: Terra e Bigonha (2008, p. 4). Texto adaptado.

Klocwork Developer: É uma das ferramentas comparadas neste trabalho. Será abordada em detalhes na subseção 2.1.1.

FindBugs : Também é outra ferramenta comparada neste trabalho. Será abordada em detalhes na subseção 2.1.2.

As ferramentas *Jlint e QJ-Pro* foram descontinuadas, ou seja, seu desenvolvimento foi interrompido.

Checkstyle é uma ferramenta de desenvolvimento para ajudar os programadores a escrever códigos Java que aderem a uma codificação padrão. Ele automatiza o processo de verificação código Java. Isto o torna ideal para projetos que pretendem impor uma codificação padrão. (CHECKSTYLE, 2008, Tradução Nossa). Ela ainda possui *plugin* para a IDE Eclipse, conforme ilustrado na Figura 1.

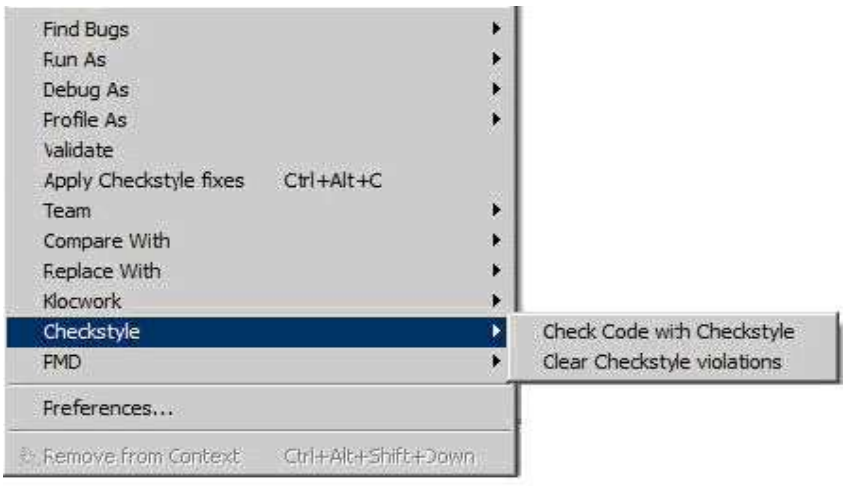


Figura 1: Módulo de extensão (plug-in) *Checkstyle* para IDE Eclipse.
Fonte: Arquivo Pessoal

PMD (2008) afirma que, *PDM* é uma ferramenta que varre códigos Java a procura de problemas como:

- Algumas anomalias, como por exemplo Bloco *try/catch/finally* vazios;
- *Dead Code* como, por exemplo, variáveis não usadas e parâmetros e métodos privados;
- Código não otimizado como, por exemplo, concatenação de *String* que é uma classe imutável ao invés de *StringBuilder* que é uma classe mutável[1];
- Expressões muito complicadas como, por exemplo, declarações desnecessárias de *if* utilização de laço *for* no lugar de *while;*
- Duplicação de código, já que, na maioria das vezes, copiar e colar código significa copiar e colar problemas.

Segundo Rutar, Almazan, Foster (2004, p. 3), *PMD* é uma ferramenta que realiza verificação sintática em código fontes. *PMD* detecta muitos *bugs* que dependem do estilo de programação. É uma ferramenta facilmente extensível por programadores, que podem escrever novos padrões de erros para detecção utilizando qualquer padrão Java ou XPath[2]. Ela ainda possui *plugin* para a IDE Eclipse, conforme ilustrado na Figura 2.

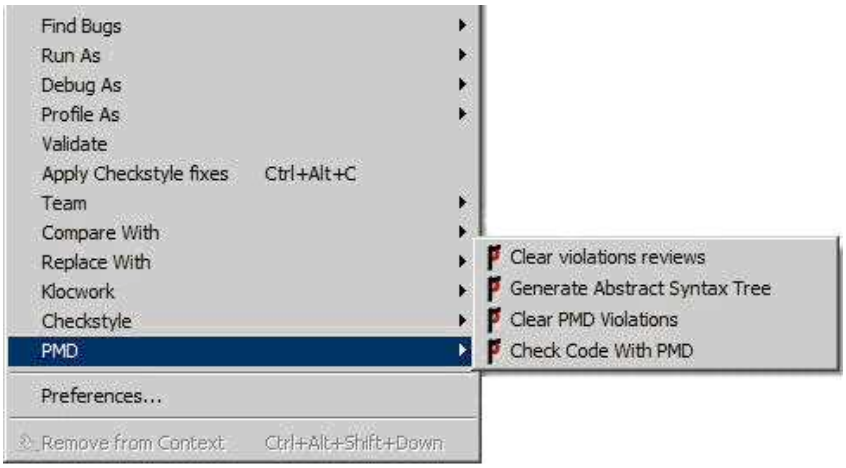


Figura 2**:** Módulo de extensão (plug-in) *PMD* para IDE Eclipse.

Fonte: Arquivo Pessoal

[1] NETO (2008) afirma que uma classe é dita imutável se o estado de um objeto da classe não pode ser alterado depois que o objeto é criado.

[2] "XPath é uma sintaxe utilizada para descrever partes de um documento XML" (TIDWELL, 2000, p. 42. Tradução e grifo nosso).

2.1.1 Klockwork Developer

Klockwork Developer (2008), afirma que, *Klockwork Developer* é uma ferramenta proprietária desenvolvida pela *Klocwork Inc.,* que é uma empresa especializada no desenvolvimento de ferramentas de análise de código fonte e para missão críticas no desenvolvimento de software. Esta empresa fornece ferramentas que permitem aos programadores identificar vulnerabilidades criticas de segurança, qualidade e defeitos arquitetônicos em C, C++ e Java.

Klocwork (2008) afirma que é uma ferramenta voltada à verificação de erros que abrange a tecnologia de análise estática. *Klockwork Developer*, como as outras ferramentas desenvolvidas pela empresa, oferece detecção automática de mais de 200 vulnerabilidades diferentes de segurança e de defeitos de qualidade. Além disso, ela ainda possui *plugin* para a IDE Eclipse, conforme ilustrado na Figura 3. O Apêndice A ilustra a aplicação do *Klocwork Developer* em um trecho de código contendo alguns erros os quais a ferramenta foi capaz de detectar.

Fisher (2007, p. 4) afirma que, embora muitos tipos de falhas podem ser encontrados utilizando a ferramenta *Klockwork Developer,* ele cita alguns exemplos para dar uma base a respeito dos erros que podem ser identificados:

- *Vulnerabilidades de segurança* - Segurança nas aplicações é cada vez mais crítico para desenvolvedores em todos os ambientes;
- *Defeitos nas implementações* - Independente da aplicação a ser desenvolvida, evitar defeitos tem impacto significativo na qualidade do produto que será desenvolvido.

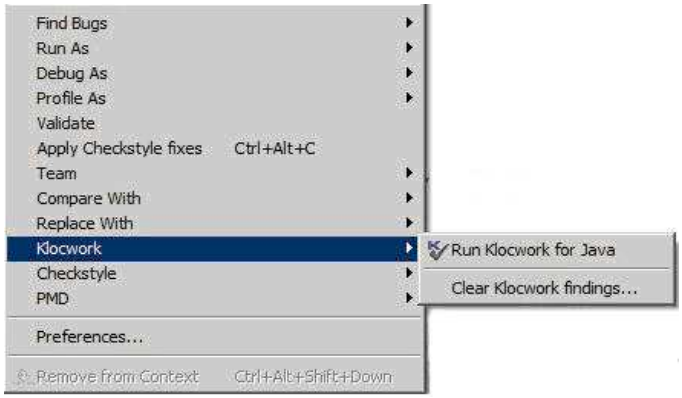


Figura 3: Módulo de extensão (plug-in) *Klocwork Developer* para IDE Eclipse.

Fonte: Arquivo Pessoal

2.1.2 FindBugs

Segundo *FindBugs* (2008), é uma ferramenta que utiliza análise estática para encontrar *bugs* em códigos Java. Ele é um software livre, distribuído sob os termos da *GNU Public License*[3] e que é desenvolvida pela *University of Marylan.* Além disso, ela ainda possui *plugin* para a IDE Eclipse, conforme ilustrado na Figura 4. O Apêndice B ilustra a aplicação do *FindBugs* em um trecho de código contendo alguns erros os quais a ferramenta foi capaz de detectar.

Rutar, Almazan, Foster (2004, p. 3) diz que, FindBugs utiliza uma série de técnicas que visam equilibrar precisão, eficiência e usabilidade. É uma ferramenta expansível, isto é, outros programadores podem definir seus próprios detectores de erros.

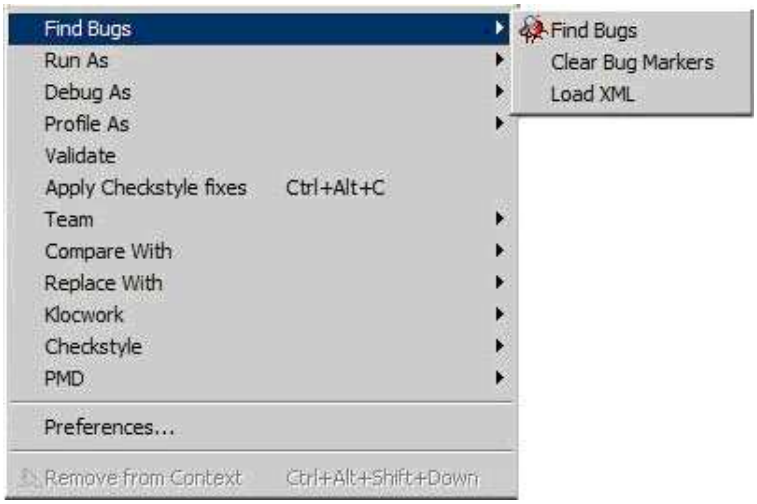


Figura 4: Módulo de extensão (plug-in) *FindBugs* para IDE Eclipse.

Fonte: Arquivo Pessoal

Grindstaff (2004, p. 1) apresenta uma lista dos mais importantes erros, apesar de que esta lista não inclui todos os problemas que a ferramenta pode encontrar:

- Valor de retorno do método ignorado. Problema ocorre quando o valor de retorno de um método é ignorado e esse não deveria ter sido;

[3] Segundo a *Free Software Fundation* (2007), o *GNU General Public License* destina-se a garantir a liberdade de compartilhar e modificar todas as versões de um software e certificando que o mesmo permanecerá livre para todos os seus utilizadores.

- Comparações com possíveis valores nulos. Problema que ocorre quando, por exemplo, algum objeto é criado recebendo o valor de um outro, que por sua vez pode estar nulo. Na seqüência, o objeto que recebeu o valor nulo é utilizado fazendo referência ao seu valor que pode ser nulo e gerar uma exceção;
- Ler atributo em um construtor antes de serem inicializados. Problema que ocorre quando atributos são lidos em construtores antes que eles sejam inicializados.

No próximo capítulo será realizado o estudo de caso aplicando-se as ferramentas em códigos com erros não identificados pelo compilador e em seguida, os resultados obtidos servirão como base para uma comparação.

3 ESTUDO DE CASO

No Capítulo 2 deste trabalho foram citadas algumas falhas que analisadores estáticos são capazes de identificar. Neste capítulo, as ferramentas serão aplicadas em exemplos de erros para que se possa realizar um estudo de caso. As duas ferramentas são extensíveis, o que significa que padrões de erros podem ser criados conforme a necessidade do desenvolvedor.

3.1 Aplicação das Ferramentas nos Erros Selecionados

3.1.1 Valor de senha visível dentro do código

Esse erro é ocasionado quando o desenvolvedor utiliza algum método que receba uma senha como parâmetro. Por exemplo, a busca de uma conexão com um banco de dados conforme ilustra a Figura 5, em que a senha de acesso ao banco está visível como um parâmetro do método. A Figura 5 ilustra o método *getConnection* que retona um tipo *Connection*. Na linha 2 dessa figura, é registrado o *driver* de conexão com o banco de dados e, na linha 3 e 4, é invocado um método que informa a *string* de conexão com o banco de dados e, então, seu resultado é retornado pelo método.

```
1 public static Connection getConnection() throws SQLException{
2    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
3    return
4         DriverManager.getConnection("jdbc:oracle:thin:@server:1521:xe",
                                      "usuario", "senha");
5 }
```

Figura 5: Exemplo de código com valor de senha visível.

Fonte: Arquivo pessoal

Quando aplica-se as ferramentas nesse código pode-se constatar que tanto o *FindBugs* quanto o *Klocwork Developer* encontram o problema na linha 4. *FindBugs* ainda alerta que qualquer usuário que tenha acesso ao código fonte pode saber a senha do banco de dados.

3.1.2 Injeção de comando SQL[4] usando classe Statement

Erro que possibilita a um usuário malicioso inserir comandos *SQL* para forçar a passagem por algum tipo de validação ou obter informações sobre o banco de dados utilizado. Um exemplo de injeção *SQL* pode ser visto na Figura 6, em que caso seja passado à seguinte *String* como *senha*: " *senha' or '1' = '1'* " o usuário conseguirá acesso ao sistema, pois o bloco condicional na linha 5 provavelmente retornará algum valor.

A Figura 6 ilustra o método *logar*, que recebe como parâmetro as *strings usuario* e *senha* e retorna se o usuário foi encontrado na base de dados. Na linha 2 dessa figura, é buscada a conexão com o SGBD e, na linha 3, é declarada uma *string* contendo a sentença SQL para a busca de um usuário pelo seu respectivo nome e senha. Na linha 4, é criada uma instrução simples com o SBGD que, na linha 5, é utilizada para executar uma consulta passando a sentença SQL. Assim, na linha 6, o método retorna se o usuário existe ou não.

```
1 public boolean logar(String usuario, String senha) throws SQLException{
2     Connection conn = ServiceConnector.getConnection();
3     String sql = "select * from user where usuario='" + usuario + "'
                                        and senha='" + senha + "'";
4     Statement stmt = conn.createStatement();
5     ResultSet rs = stmt.executeQuery(sql);
6     return rs.next();
7 }
```

Figura 6: Exemplo de código com vulnerável a injeção SQL.

Fonte: Terra (2008b) Texto adaptado

[4] Santos (2007) define *SQL injection* como uma vulnerabilidade de segurança que ocorre na camada de dados da aplicação. A vulnerabilidade é presente quando a entrada de dados para o usuário não filtra corretamente, dentro do que foi digitado, comandos SQL.

Aplicando-se as ferramentas no código constatou-se que *FindBugs* e *Klocwork Developer* identificam a falha e informam que nesse caso o mais seguro seria utilizar a classe *PreparedStatement*[5].

3.1.3 Conexão com banco de dados aberta e não fechada

Esse erro é caracterizado quando uma conexão com um banco de dados é realizada e no término da transação ela não é fechada. Um exemplo desse erro é ilustrado na Figura 7, em que é aberta uma conexão com um banco de dados na linha 2 dentro do método *inserir* e, em seguida, é executado uma inserção de valores em uma tabela desse banco, porém, ao término do método, a conexão com o banco de dados não é encerrada.

A Figura 7 ilustra o método *inserir* que recebe um objeto da classe *Contato* como parâmetro. Na linha 2 dessa figura, é buscada a conexão com o SGBD e, nas linhas 3 a 7, é realizado a inserção do contato, a qual é confirmada, através do método *commit*, na linha 8.

```
1 public void inserir(Contato c) throws SQLException{
2     Connection conn = ServiceConnector.getConnection();
3     PreparedStatement pst =
            conn.prepareStatement("insert into contatos values (?,?)");
4     pst.setInt(1,c.getId());
5     pst.setString(2,c.getNome());
6     pst.executeUpdate();
7     pst.close()
8     conn.commit();
9 }
```

Figura 7: Exemplo de conexão com banco de dados aberta e não encerrada.

Fonte: Arquivo Pessoal

[5] Hewitt (2003) define PreparedStatement como uma classe que permite executar uma consulta precompilada de maneira mais eficiente. É utilizada para realizar consultas parametrizadas.

Ao aplicar-se as ferramentas no código pode-se constatar que o *FindBugs* e o *Klocwork Developer* alertam sobre a falha existente nesse método. O *FindBugs* informa que uma conexão não encerrada com o banco de dados pode resultar em mau desempenho e ainda causar problemas de comunicação com o banco de dados. Já o *Klocwork Developer* afirma que essa falha pode ativar várias exceções.

3.1.4 Valor de retorno do método ignorado

Esse erro é comumente ocasionado quando um método, de um objeto imutável, é com a intenção de alterá-lo. Um exemplo comum é o da classe *java.lang.String* em que o retorno do método *toLowerCase,* que retorna a *string* em minúsculo, é ignorado e deixa a impressão que a *string* foi convertida para minúsculo, conforme pode ser observado na Figura 8.

```
1 public static void imprimeMinusculo(String nome) {
2     nome.toLowerCase();
3     System.out.println(nome);
4 }
```

Figura 8: Demonstração de valor de retorno de método ignorado.
Fonte: Arquivo pessoal

Aplicando-se as ferramentas nesse código verificou-se que tanto o *FindBugs* quanto o *Klocwork Developer* localizam o problema na linha 2.

3.1.5 Utilização de atributos em um construtor antes de serem inicializados

Anomalia que ocorre quando um desenvolvedor executa, dentro do construtor do objeto, a leitura de um atributo que não foi inicializado. Esse erro não foi alertado

pelo compilador, pois, em Java, os atributos do tipo *Object* de uma classe são inicializados com valor nulo no momento da construção do objeto. Um exemplo pode ser observado na Figura 9, em que o atributo *lista* da classe não é inicializado em sua declaração e mesmo assim é utilizado pelo construtor da classe.

A Figura 9 possui ilustra a classe *Demonstracao*. Essa classe possui, na linha 2, um atributo do tipo *List* chamado *lista* e, nas linhas 3 a 7, é criado o construtor da classe que invoca o método *add* do atributo *lista*, porém o mesmo não foi previamente inicializado.

```
1 public class Demonstracao {
2     private List<String> lista;
3     public Demonstracao(String valorInicial){
4           StringTokenizer tok = new StringTokenizer(valorInicial);
5           while (tok.hasMoreTokens()){
6                 lista.add(tok.nextToken());
7           }
8     }
9 }
```

Figura 9: Demonstração de código com leitura de atributo antes de sua inicialização.

Fonte: Arquivo pessoal.

Ao se realizar uma análise no código utilizando as ferramentas foi constatado que o *FindBugs* encontra o problema e alerta destacando que o atributo não havia sido inicializado e sugere que o mesmo seja inicializado. Já o *Klocwork Developer* não identifica a anomalia no código.

3.1.6 Utilização do método equals para comparar um vetor com um objeto que não é vetor

Esse erro ocorre quando o valor de um vetor é comparado com algum objeto que não seja vetor. Esse tipo de comparação sempre retornará falso, pois o que está sendo comparado são objetos diferentes. Um exemplo desse erro é ilustrado na Figura 10, em que, na linha 2, o vetor de *string v* é comparado com uma *string* vazia.

```
1 public boolean vazio(String[] v) {
2       return v.equals("");
3 }
```

Figura 10: Exemplo de código com comparação de objeto vetor com objetos não vetores.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando o *FindBugs* é retornado um alerta que diz respeito a comparação de vetor com um objeto de tipo diferente do vetor, ele ainda destaca que essa comparação sempre retornará falso e que caso deseje comparar o conteúdo de dois vetores deve-se usar o método *equals* da classe *java.util.Arrays*. Já o *Klocwork Developer* alerta sobre o problema, também destacando que a comparação sempre retornará falso, porém, não indica a melhor forma de executar a comparação, somente destaca que, para comparações de vetores, deve-se comparar os seus elementos.

3.1.7 Chamada do método equals comparando objetos de tipos diferentes

Esse erro ocorre quando o método *equals* de um objeto é invocado e o objeto utilizado como parâmetro de comparação não são do mesmo tipo. Um exemplo desse erro pode ser observado na Figura 11, que, na linha 2, uma variável do tipo *String* é comparada com uma variável do tipo *StringBuilder*. O correto seria realizar uma comparação da variável do tipo *String* com o valor retornado pelo método *toString* da variável do tipo *StringBuilder*.

```
1 public boolean verificaIgualdade(String str, StringBuilder strBuilder){
2     return strBuilder.equals(str);
3 }
```

Figura 11: Exemplo de comparação de objetos diferentes através no método *equals.*

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código tanto o *FindBugs* quanto o *Klocwork Developer* identificaram o erro quando aplicadas na classe mostrada na Figura 11. *FindBugs* ainda informa que provavelmente os objetos comparados não são originados da mesma classe e que o resultado dessa comparação será sempre falso.

3.1.8 Atributo de uma classe não utilizado

Problema caracterizado quando um atributo é declarado em uma classe e esse nunca é usado. Um exemplo desse erro pode ser observado na Figura 12, em que um atributo é declarado na linha 2 e inicializado no construtor da classe como pode ser visto na linha 5. No entanto, não é utilizado em nenhum dos métodos da classe.

```
1 public class AtributoNaoUsado {
2     int atributo;
3     public AtributoNaoUsado() {
4           super();
5           atributo = 15;
6     }
7     public void metodo1() {...}
8     public void metodo2() {...}
9 }
```

Figura 12: Exemplo de código com atributo nunca usado.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando o *Klocwork Developer* não houve alerta sobre o problema, pois, pelo fato do atributo possuir visibilidade *default,* ele considera que o atributo pode ser usado em outra classe do pacote, enquanto que o *FindBugs* alerta sobre a falha na linha 5 e sugere que o mesmo seja removido do código.

3.1.9 Container adicionado a ele mesmo

Problema gerado quando algum *container* é adicionado a ele próprio. O que pode gerar uma exceção de *Overflow*[6]*.* Um exemplo desse problema pode ser observado na linha 2 da Figura 13, na qual a variável de coleção *list* é adicionada a ela mesma.

```
1 public void container(List<Object> list) {
2     list.add(list);
3 }
```

Figura 13: Exemplo de container adicionado a ele mesmo.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando o *Klocwork Developer* não houve alerta sobre o problema, enquanto que *FindBugs* informa sobre o problema alertando que o mesmo pode causar uma exceção do tipo *StackOverflowException*[5]*.*

3.1.10 Loop infinito em laço de repetição

Erro decorrente de algum laço de repetição sem condição de saída. Um exemplo desse problema pode ser observado na Figura 14, em que o bloco de repetição *while*, na linha 3, termina quando a variável *j* for maior que 20, porém, dentro do bloco *while,* a variável nunca é incrementada, nem sequer alterada.

[6] Segundo Sun Microsystem Inc. (2008) é uma exceção acionada quando uma pilha estoura devido a vários pedidos recursivos.

```
1 public void metodo() {
2       int n=0; int j=0;
3       while(j<20){
4             /*processamento do laço sem condição de parada*/
5             n++;
6       }
7 }
```

Figura 14: Exemplo de *loop* infinito.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando o *FindBugs* houve alerta sobre o erro de *loop* infinito alertando que esse erro pode ativar uma exceção, a ferramenta *FindBugs* não informa o tipo de exceção. Já o *Klocwork Developer* não identifica o problema.

3.1.11 Loop infinito em recursão

Esse *loop* infinito é causado pela chamada recursiva de um método. Um exemplo desse tipo de *loop* pode ser observado na Figura 15, em que o método *metodoRecursivo* chama a ele mesmo, sem nenhuma condição de parada.

```
1 public void metodoRecursivo() {
2     /* Processamento do método sem condição de parada*/
3     metodoRecursivo();
4 }
```

Figura 15: Exemplo de *loop* infinito em recursão.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando as ferramentas ambas alertaram sobre o problema de chamada recursiva realizada pelo método *metodoRecursivo*. O *Klocwork Developer* informa que esse erro fará com que a JVM retorne um erro *StackOverflowError.* O *FindBugs* alerta que, incondicionalmente, esse método invoca a si próprio e que pode resultar em uma exceção de *StackOverflowError.*

3.1.12 Loop infinito métodos invocados mutuamente

Esse erro é decorrente da chamada de dois ou mais métodos entre si de forma recursiva e também sem condição de saída em todos métodos invocados. Um exemplo desse erro pode ser observado nas linhas 4 e 8 da Figura 16, na qual os métodos existentes, *metodoA* e *metodoB*, se invocam mutuamente sem nenhuma condição de parada em qualquer um dos métodos.

```
1 public class ExemploLoop {
2     public void metodoA() {
3           /* Processamento do método sem condição de parada */
4           metodoB();
5     }
6     public void metodoB() {
7           /* Processamento do método sem condição de parada */
8           metodoA();
9     }
10}
```

Figura 16: Exemplo de *loop* infinito.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando o *FindBugs* e o *Klocwork Developer* não foi identificado o problema de *loop* causado pelos métodos que possuem chamada mútuas e sem condição de parada.

3.1.13 Verificação de valor nulo redundante

Problema ocorrido quando testa-se o valor de um objeto para verificar sua nulidade já se sabendo que o mesmo é nulo. Exemplo desse problema pode ser verificado na Figura 17, em que, na linha 2, o parâmetro *valor* do método é testado se é diferente de nulo e, na linha 3, é adicionado ao objeto *lista*. Mesmo assim, na

linha 4, é novamente verificado se é diferente de nulo para que possa ser dada a saída de seu valor.

```
1 public void metodo(List<String> lista, String valor) {
2     if (lista != null && valor != null){
3           lista.add(valor);
4           if (valor != null) {
5                System.out.print("Inserido valor "+ valor +"na lista");
6           }
7     }
8 }
```

Figura 17: Exemplo de código com verificação de nulo redundante.

Fonte: Arquivo Pessoal

Ao se realizar uma análise no código utilizando as ferramentas pode-se constatar que o *Klocwork Developer* e o *FindBugs* alertaram sobre a redundância existente na comparação da nulidade do objeto *valor*. As ferramentas ainda destacaram que o valor do objeto já é conhecido, isto é, a garantia que seu valor é diferente de nulo já foi constatada no bloco condicional anterior.

3.1.14 Stream não encerrada

Esse erro ocorre quando um objeto da classe *Stream* é instanciado, utilizado e, ao término do código, não é encerrado. Exemplos comuns desse erro ocorrem na manipulação de arquivos, conforme pode ser observado na Figura 18. No método ilustrado nessa figura, é criada, na linha 4, um fluxo de escrita em um dado arquivo (linha 2) e, na linha 5, é escrito seu conteúdo. No entanto, em nenhum ponto desse método, a *stream* aberta é encerrada, o que possivelmente gerará alguma falha na gravação dos dados.

```
1 public void grava(String nomeArquivo, int conteudo){
2     try{
3           File file = new File(nomeArquivo);
4           OutputStream out = new FileOutputStream(file,false);
5           out.write(conteudo);
6     }catch (IOException e){
7           //Mensagem de retorno da exceção
8     }
9 }
```

Figura 18: Exemplo de não fechamento de objeto do tipo *FileInputStream.*

Fonte: Terra (2008a) Texto Adaptado.

Ao aplicar-se as ferramentas constatou-se que tanto o *Klocwork Developer* quanto o *FindBugs* alertam sobre a falha existente. O *Klocwork Developer* informa que esse tipo de falha poder gerar várias exceções. O *FindBugs* informa que quando se abre um *stream* deve-se usar o bloco *try/catch/finally* para tratamento de exceções e o método *close* para garantir que a *stream* aberta será encerrada.

3.1.15 Bloco catch vazio

Erro que ocorre quando é utilizado um bloco *try*/*catch* para tratamento de exceções e o bloco *catch,* que é a parte responsável pelo retorno caso haja alguma exceção, está vazio. Um exemplo desse erro pode ser observado na Figura 19, em que, na linha 4, o método *formataData* retorna o resultado no método *parse* de um objeto do tipo *DateFormat*, que pode ativar uma exceção do tipo *ParseException.* Porém, o método não provê tratamento caso ocorra essa exceção, isto é, o bloco de tratamento está vazio (linha 5 e 6).

```
1 public Date formataData(String strData) {
2     DateFormat df = new SimpleDateFormat("dd/MM/yyyy");
3     try{
4           return df.parse(strData);
5     }catch(ParseException e){
6     }
7     return null;
8 }
```

Figura 19: Exemplo de bloco *catch* vazio.

Fonte: Arquivo Pessoal

Ao aplicar-se as ferramentas em um código com esse erro pode-se constatar que o *Klocwork Developer* alerta sobre o problema e que o *FindBugs* não identifica a falha. O *Klocwork Developer* ainda destaca que, se ocorrer uma exceção no código, o certo seria tratá-la e não ignorá-la.

3.1.16 Comparação de variável com ela mesma

Erro decorrente de casos em que uma variável é comparada com ela mesma. Um exemplo desse erro é ilustrado na Figura 20, em que no método *verifica* são declarados dois objeto da classe *Date* e o valor de um dos objetos é atribuído ao outro (linha 3), o que significa que os dois referenciam ao mesmo objeto. No entanto, já na linha 4, existe uma comparação entre os valores desses objetos que sempre retornará verdadeiro.

```
1 public boolean verifica() {
2     Date data = new Date();
3     Date data2 = data;
4     if (data==data2){
5           return true;
6     }
7     return false;
8 }
```

Figura 20: Exemplo de comparação de variável com ela mesma.

Fonte: Arquivo Pessoal

Ao aplicar-se as ferramentas no código. O *FindBugs* e *o Klocwork Developer* identificaram a falha no trecho de código. *FindBugs* sugere que seja verificado o que está sendo comparado e afirma que o erro pode ser lógico. *Klocwork Developer* alerta sobre a anomalia e diz que isso pode indicar um erro no código.

3.2 Análise Comparativa dos Resultados

A Tabela 2 exibe o resultado da aplicação das ferramentas nos erros da seção 3.1. O símbolo "✓" significa que o erro foi identificado e o símbolo "**X**" significa que a ferramenta não identificou o erro.

Erros	***FindBugs***	***Klocwork Developer***
Valor de senha visível dentro do código (subseção 3.1.1)	✓	✓
Injeção de comando *SQL* usando classe *Statement* (subseção 3.1.2)	✓	✓
Conexão com banco de dados aberta e não fechada (subseção 3.1.3)	✓	✓
Valor de retorno do método ignorado (subseção 3.1.4)	✓	✓
Utilização de atributos em um construtor antes de serem inicializados (subseção 3.1.5)	✓	**X**
Utilização do método *equals* para comparar um vetor com um objeto que não é vetor (subseção 3.1.6)	✓	✓
Chamada do método *equals* comparando objetos de tipos diferentes (subseção 3.1.7)	✓	✓
Atributo de uma classe não utilizado (subseção 3.1.8)	✓	**X**
Container adicionado a ele mesmo (subseção 3.1.9)	✓	**X**
Loop infinito em laço de repetição (subseção 3.1.10)	✓	**X**
Loop infinito em recursão *(subseção 3.1.11)*	✓	✓
Loop infinito métodos invocados mutuamente (subseção 3.1.12)	**X**	**X**
Verificação de valor nulo redundante (subseção 3.1.13)	✓	✓
Stream não encerrada (subseção 3.1.14)	✓	✓
Bloco *catch* vazio (subseção 3.1.15)	**X**	✓
Comparação de variável com ela mesma (subseção 3.1.16)	✓	✓
Total	**14/16 (87,5%)**	**11/16 (68,75%)**

Tabela 2: Resultado dos testes realizados com as ferramentas de AEC.
Fonte: Arquivo Pessoal

A Tabela 2 mostra que o FindBugs foi mais eficaz que o Klocwork Developer. Dos 16 erros testados, o FindBugs encontrou 14 erros (87,5%), enquanto que o Klocwork Developer encontrou 11 erros (68,75%).

Porém, para se medir a eficácia e eficiência das ferramentas deve-se levar em consideração os erros identificados e os não identificados, verificando sua integridade e consistência[7]. Além disso, pelo fato das ferramentas serem extensíveis, o desenvolvedor pode criar novos padrões de erros e manter a ferramenta eficaz para sua necessidade. O que leva a conclusão que as ferramentas por si só não são capazes de identificar todos os erros sem a análise humano e não se pode afirmar qual delas é a melhor.

[7] Segundo Ball e Rajamani (2002), integridade é se cada erro relatado é um erro real e consistente se todo erro real é relatado pela ferramenta.

4 CONSIDERAÇÕES FINAIS

Atualmente, é evidente a valorização da qualidade em programação. Pádua (2003, p.7) afirma que geralmente a qualidade de um produto decorre diretamente da qualidade do processo utilizado na produção dele.

Segundo Pressman (2005, p. 786), a atividade de teste de software é um elemento crítico da garantia de qualidade da construção de um programa e, por isso, é altamente viável a aplicação de técnicas de inspeção de código fonte. Segundo Sommerville (2007, p. 345), essas técnicas surgiram com o objetivo de detectar defeitos de programa antes de sua implantação, idéia que vem desde a década de 1970, formalizada pela primeira vez pela IBM.

Ainda segundo Sommerville (2007, p. 345), analisadores estáticos são ferramentas de software que varrem o código fonte de um programa e detectam possíveis defeitos e anomalias. Atuam de forma a complementar os recursos de detecção de erros providos pelo compilador da linguagem. AEC é uma das técnicas de Inspeção de Software no processo de V&V.

Ao realizar AEC, mais precisamente em Java, pretende-se que o resultado seja consistente e íntegro, isto é, reporte todos os erros existentes e todos os erros reportados sejam realmente erros.

Braz (2007) afirma que, as ferramentas comerciais existentes se fundamentam no mesmo princípio: regras e padrões de codificação suspeitos. Por isso as tornam muito dependentes da análise humana.

Ele ainda afirma que os principais problemas das ferramentas de análise de código fonte estão concentrados em:

- **Falso negativo**: O programa contém erros não alertados pela ferramenta. Isso dá a falsa sensação da não existência de erros, mas, na verdade, apenas significa que a ferramenta não foi capaz de encontrar outros tipos de erros.
- **Falso positivo**: A ferramenta alerta trechos normais de código como erros. Isso ocorre por dois motivos: um erro catalogado pela ferramenta que na verdade não existe ou uma classificação de erro incoerente com o

ambiente em questão. Por exemplo, um *bug* de *SQL Injection* identificado em um software que não possui interesse nesse tipo de *bug* devido as suas características de operação.

De Acordo com o estudo de caso abordado, cujo resultado sintetizado pode ser observado na Tabela 3, conclui-se que *FindBugs* foi mais eficaz do que *Klocwork Developer* e seria a melhor opção para realizar AEC em Java. Um outro ponto positivo na ferramenta *FindBugs* é o fato da ferramenta ser gratuita.

	Ferramentas	
	FindBugs	*Klocwork Developer*
Percentual de erros	14/16 (87,5%)	11/16 (68,75%)

Tabela 3: Resumo do resultado do Estudo de Caso.
Fonte: Arquivo pessoal

Porém, como já foi mencionado, as ferramentas são extensíveis o que permite a criação de novos padrões de erros para detecção deixando-as assim eficazes para o seu propósito. Esse fato leva a conclusão de que as ferramentas por si só não são capazes de identificar todos os erros sem o acompanhamento humano e não se pode afirmar categoricamente qual é a melhor ferramenta de análise estática de código, contudo estudos podem ser realizados visando mensurar suas integridades e consistências.

Contudo, Sommerville (2007) afirma que os testes podem mostrar somente a presença de erros em um programa, ou seja, eles não podem demostrar que não existam defeitos remanescentes. Portanto, testes nunca podem assegurar que um programa está correto, simplesmente convencem seus desenvolvedores e clientes que o programa está confiável o suficiente para a utilização. Reforçando isso, Louridas (2006) também afirma que nenhum analisador de código pode garantir que o programa está correto, tal garantia é impossível.

REFERÊNCIAS

BALL, T.; RAJAMANI, S. K. *The slam project: Debugging system software via static analysis*. In: *Proceedings of 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Vancouver: ACM, 2002.

BRAZ, F. *Ferramentas de Análise Código Fonte* Disponível em: < http://softwares eguro.blogspot.com/2007/07/ferramentas-de-anlise-cdigo-fonte.html>. Acesso em: 17 nov. 2008.

CHECKSTYLE, *Checkstyle*. 2008 Disponível em: <http://checkstyle.sourceforge.net/> . Acesso em: 17 nov. 2008.

FINDBUGS, *FindBugs – Find Bugs in Java Programs*. 2008. Disponível em: <http://findbugs.sourceforce.net/index.html>. Acesso em: 17 nov. 2008.

GRINDSTAFF, C. *FindBugs, part 1: Improve the quality of your code.* Disponível em: <http://www.ibm.com/developerworks/java/library/j-findbug1/>. Acesso em: 13 out. 2008.

HEWITT, E. **Java™ for ColdFusion® Developers.** Pretience Hall, 2003.

KALINOWSKI, M. *Engenharia de Software - Introdução à Inspeção de Software*. Disponível em: <http://www.devmedia.com.br/articles/viewcomp.asp?co mp=8037>. Acesso em: 11 set. 2008.

LOURIDAS, P. *Static Code Analysis. IEEE Software.* P 58-61, Jul 2006.

KLOCWORK, *klocwork Developer for Java* Disponível em: <http://www.klocwork.com/products/developerJava.a sp>. Acesso em: 17 nov. 2008.

NETO, Domingos. *Immutable Classes*, 2008. Disponível em: <http://www.codeinstructions.com/2008/07/immutable-classes.html>. Acesso em: 22 nov. 2008.

PÁDUA, W. **Engenharia de Software** 2. ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora S.A., 2003. Caps. 1, p 7.
PMD. *PMD*. 2008. Disponível em: <http://pmd.sourceforge.net/>. Acesso em: 17 nov. 2008.

PRESSMAN, R. S. **Engenharia de Software** 5. ed. São Paulo: Pearson Makron Books, 2005. Caps. 18-19, p. 786 e 836.

RUTAR, N, ALMAZAN, C. B. , FOSTER, J. S. *A Comparison of Bug Finding Tools for Java.* Artigo, 15º ISSRE (International Symposium on SoftwareReliability Engineering). Nov 2004. Disponível em: <http://www.cs.umd.edu/~jfoster/papers/issre04.pdf>. Acesso em: 13 out. 2008.

SANTOS, A. L. dos. **Integração de Sistemas com Java**. São Paulo: Brasport, 2007.

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. São Paulo: Pearson Addison-Wesley, 2007. Caps. 22-23, p. 341-354.

SUN MICROSYSTEM INC. *Java™ Platform, Standard Edition 6 API Specification* .2008. Disponível em: <http://java.sun.com/javase/6/docs/api/>. Acesso em: 21 nov. 2008.

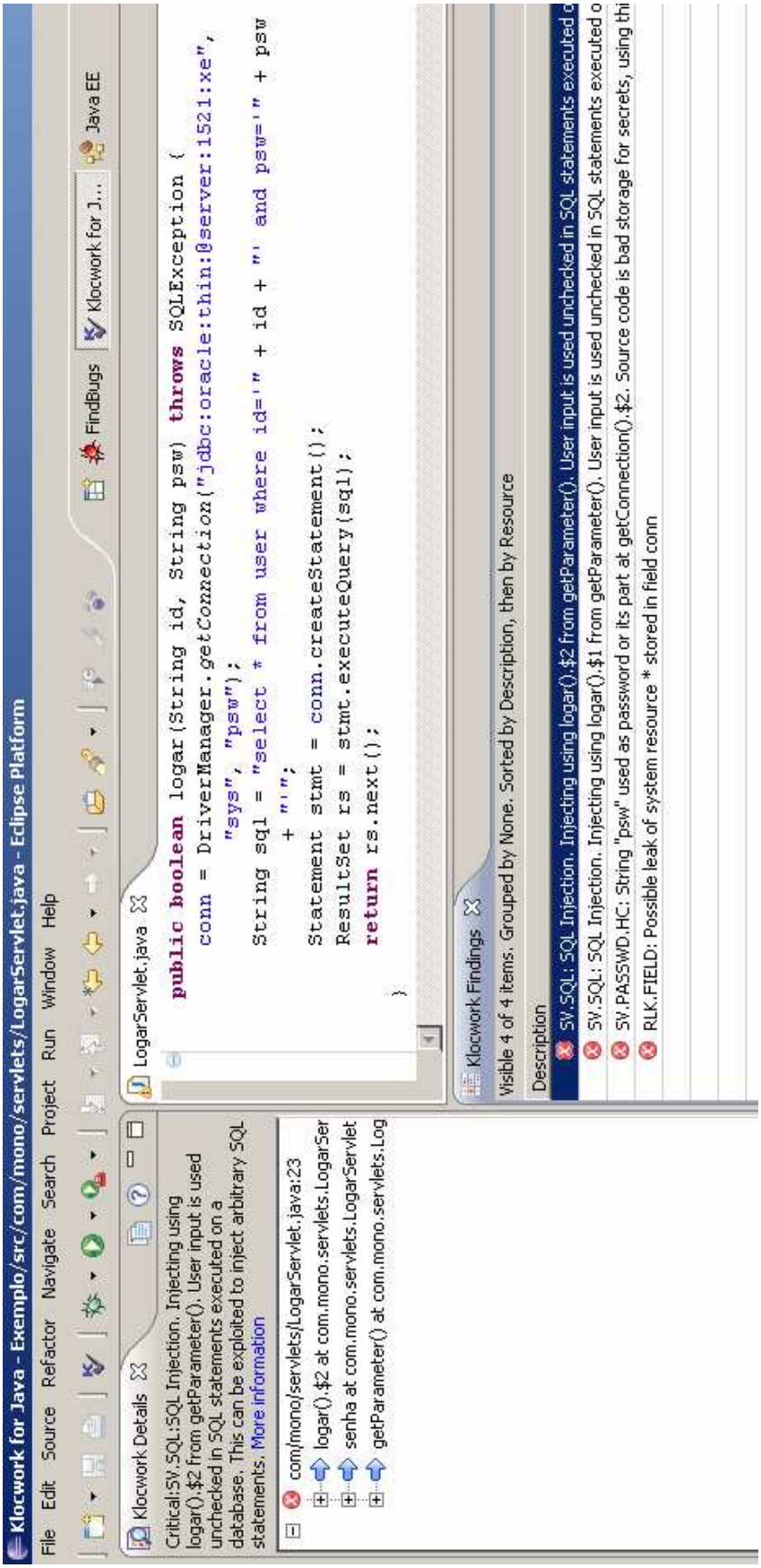
TERRA, Ricardo, BIGONHA, R. S. *Ferramentas para Análise Estática de Códigos Java.* Artigo, EBTS (Encontro Brasileiro de Teste de Software). Out. 2008.

TERRA, Ricardo. *Java SE Manipulação de Streams* . Belo Horizonte 2008a. Disponível em: < http://www.ricardoterra.com.br/faminas/>. Acesso em: 21 nov. 2008

TERRA, Ricardo. *Java SE Trabalhando com Banco de Dados*. Belo Horizonte 2008b. Disponível em: < http://www.ricardoterra.com.br/faminas/>. Acesso em: 21 nov. 2008.

TIDWELL, Doug. **XSLT**. Sebastopol: O'Reilly Media, 2001.

APÊNDICE A – Perspectiva do *Klocwork Developer* na IDE Eclipse



APÊNDICE B – Perspectiva do *FindBugs* na IDE Eclipse