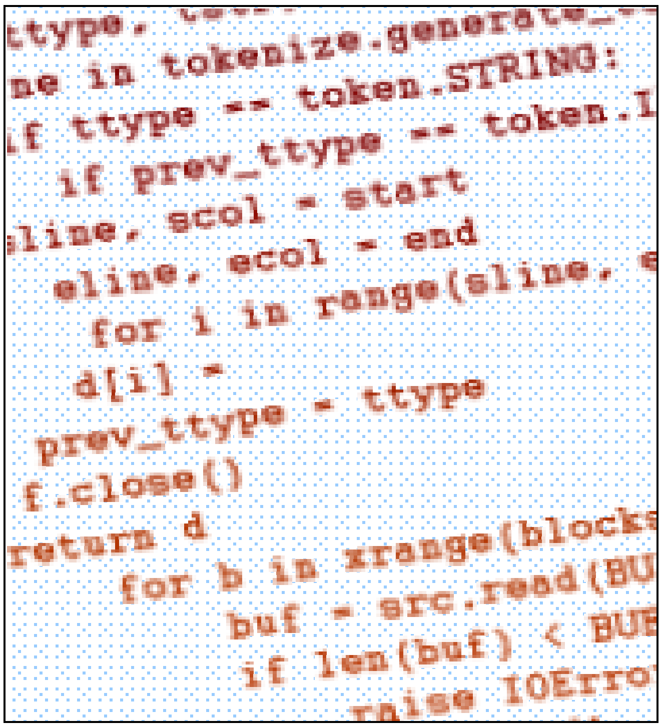
Advance Topics in Software Engineering

Contribution to the Open Source Project,
Checkstyle

by Ioannis Sermetziadis
(iserm@dmst.aueb.gr)



Professor:
Diomidis Spinellis

Athens University of Economics and Business
Department of Management Science and Technology
Athens, 2006

Table of Contents

Table of Contents **2**

1. Introduction: My decision to contribute to Checkstyle **3**

2. Project Description **3**

2.1 Checkstyle's structure: How it works **4**

2.2 Checkstyle's GUI **5**

3. My contribution to the Checkstyle **6**

3.1 New check, ClassblanknessCheck.java **6**

3.2 Improvement of RedundantImportCheck.java functionality **6**

3.3 New properties of StrictDuplicateCodeCheck **7**

3.4 A new reporting method, JavaLogger **7**

3.5 Changes to Checkstyle's GUI **8**

4. APPENDIX: **10**

4.1 Source code: **10**

ClassBlanknessCheck.java 10

Part of RedundantImportCheck.java, where I added code 13

Code I added in StrictDuplicateCodeCheck.java 13

JavaLogger.java 14

Part of Main.java, where I added code 17

Part of ParseTreeInfoPanel.java, where I added code 18

4.2 Contact with the project's administrator **19**

I wrote: 19

Oliver Burn wrote: 20

4.3 Track upload and administrator's comment: **21**

1. Introduction: My decision to contribute to Checkstyle

After looking around in sourceforge.net for a long period, I had many ideas of some projects that I could contribute to, such as text editors, VoIP projects, sound streaming projects and many more. After this search, there are many projects that I have included in my everyday use, like ant, vim and 7-zip. When the lessons of the course, Advance Topics in Software Engineering, started, I had the chance to be in a lab lesson, and this was when I had my first experience with some Unix tools. Grep was after on always in my mind as a genius program, mainly cause many times before I was trying to find a short string among many text files, and this task was something that always was really time consuming. Of course, contributing to a tool like Grep sounded as something really difficult mainly because of the programming language restriction, as the only language I have been taught and I can fluently use is Java. I started looking for a similar project implemented in Java and Checkstyle appeared in my way.

Checking the style of Java code, is something really interesting and useful. Many Open Source Projects have been implemented in Java, and need to be kept in some coding standards, something that is really hard as many people from all around the world write and contribute code to these projects. As checkstyle is implemented in Java, it is OS independent.

2. Project Description

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. By default it supports the Sun Code Conventions, but is highly configurable. It can be invoked with an ANT task and a command line program. It is registered under GNU Library or Lesser General Public License (LGPL), it consists of five main developers and Oliver Burn is the main administrator of the project. It was registered in sourceforge.net in the 20th of June in 2001 and 4.1 version in out since the 19th of December 2005. It is in the mature stage now. At first, it was intended to check code layout issues, but since the internal architecture was changed in version 3, more and more checks for other purposes have been added. Some of these new checks have to do with javadoc comments, naming conventions, headers, imports, size violations, whitespace, modifiers, block checks, coding, class design, duplicate code and metrics. There have also been written plug-ins, so that checkstyle can be used from inside a developing environment. The project‘s implementation package consists of 268 java files or 55.596 lines of code or 1.809.547 bytes. The whole checkstyle’s package, that also contains the JUnit tests and that these tests use consists of 599 java files or 77824 lines of code or 2.502.288 bytes.

2.1 Checkstyle’s structure: How it works

Checkstyle consists of three main modules: a) *FileSetChecks* that take a set of input files and fire error messages, b) *Filters* that filter audit events, including error messages, for acceptance and c) *AuditListeners*, which report accepted events. Modules are structured in a tree whose root is the *Checker* module. Many checks are submodules of the *TreeWalker,* which implements FileSetCheck module(interface). The TreeWalker operates by separately transforming each of the Java source files into an abstract syntax tree and then handing the result over to each of its submodules which in turn have a look at certain aspects of the tree. Checkstyle obtains a configuration from an XML document whose elements specify the configuration's hierarchy of modules and their properties. When using this tool from the command line you call the Main class of the default package of the project *“com.puppycrawl.tools.checkstyle”*. This class takes the tool’s arguments and calls the corresponding classes , to implement the checks mentioned in the configuration ‘xml’ file which is required when using the tool. Here are the arguments that can be given when calling checkstyle:

-c configurationFile.xml

-n packageNamesFile - specify a package names file to use.

-f format - specify the output format. Options are "plain" for DefaultLogger.java, "xml" for XMLLogger.java and "java" for JavaLogger.java (one of my inserts)

-p propertiesFile - specify a properties file to use.

-o file - specify the file to output to.

-r dir - specify the directory to traverse for Java source files.

Example of running the programme from the command line:
java com.puppycrawl.tools.checkstyle.Main -c sun_checks.xml -f plain -o output.txt -r inputDir

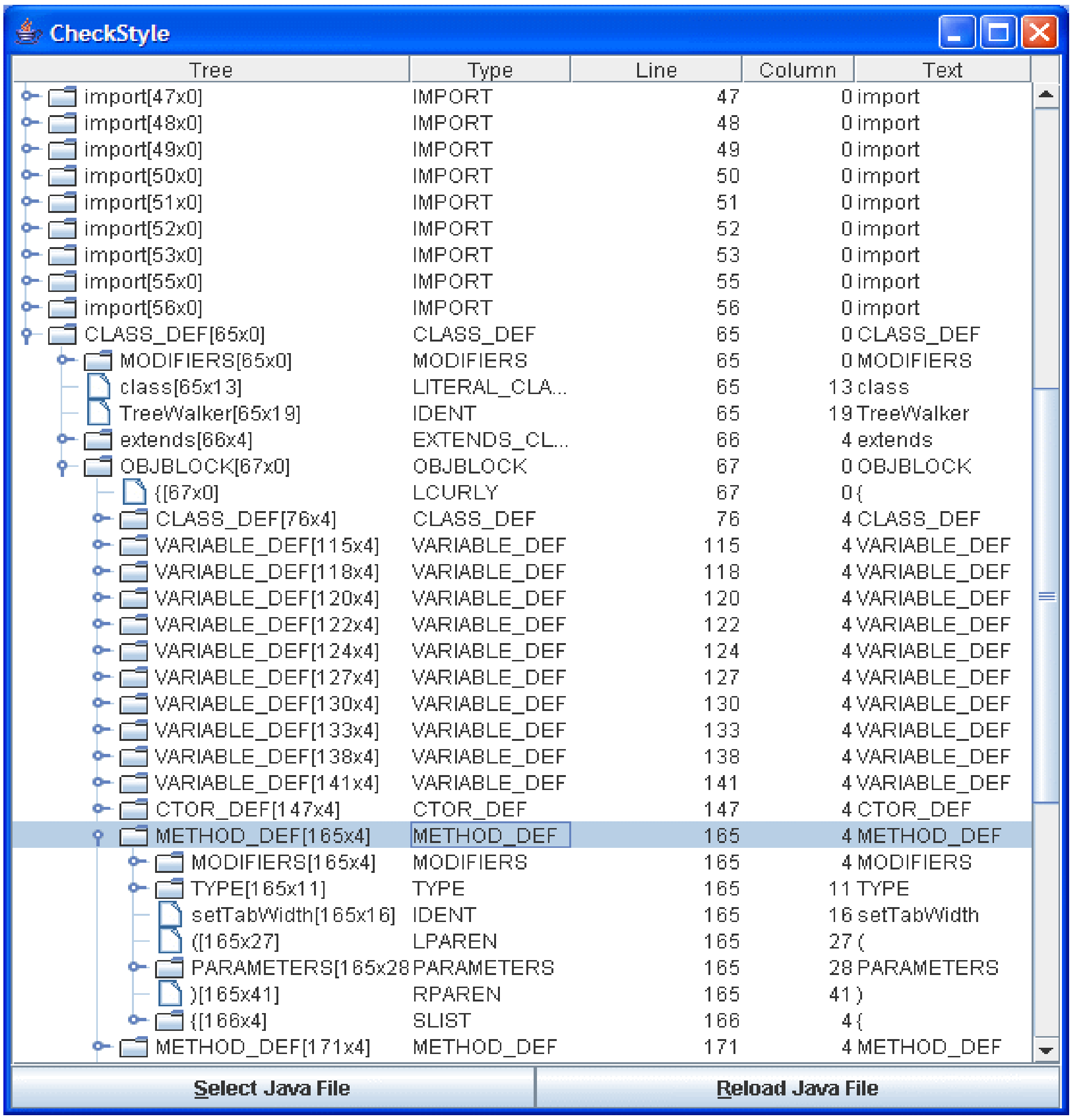
More details of how Checkstyle works can be found on the project’s page in sourceforge.net: http://checkstyle.sourceforge.net

2.2 <u>Checkstyle's GUI</u>

Checkstyle's Gui is based on Java grammar and structure. It is used to represent a java file's structure which is generated through the ANTLR parser. This parser is also used by TreeWalker when it is asked to implement some checks. By using this tool, you can select a syntactically correct Java source file, and ANTLR can transform a stream of characters (a Java file) into a tree representation that reflects the structure of the file.This command can be used from the command line to call Checkstyle's GUI:

java -classpath checkstyle-all-4.1.jar com.puppycrawl.tools.checkstyle.gui.Main

As you navigate from the root of the tree to one of the leafs, you'll notice that the token type denotes smaller and smaller units of your source file.



Tree	Type	Line	Column	Text
import[47x0]	IMPORT	47	0	import
import[48x0]	IMPORT	48	0	import
import[49x0]	IMPORT	49	0	import
import[50x0]	IMPORT	50	0	import
import[51x0]	IMPORT	51	0	import
import[52x0]	IMPORT	52	0	import
import[53x0]	IMPORT	53	0	import
import[55x0]	IMPORT	55	0	import
import[56x0]	IMPORT	56	0	import
CLASS_DEF[65x0]	CLASS_DEF	65	0	CLASS_DEF
MODIFIERS[65x0]	MODIFIERS	65	0	MODIFIERS
class[65x13]	LITERAL_CLA...	65	13	class
TreeWalker[65x19]	IDENT	65	19	TreeWalker
extends[66x4]	EXTENDS_CL...	66	4	extends
OBJBLOCK[67x0]	OBJBLOCK	67	0	OBJBLOCK
{[67x0]	LCURLY	67	0	{
CLASS_DEF[76x4]	CLASS_DEF	76	4	CLASS_DEF
VARIABLE_DEF[115x4]	VARIABLE_DEF	115	4	VARIABLE_DEF
VARIABLE_DEF[118x4]	VARIABLE_DEF	118	4	VARIABLE_DEF
VARIABLE_DEF[120x4]	VARIABLE_DEF	120	4	VARIABLE_DEF
VARIABLE_DEF[122x4]	VARIABLE_DEF	122	4	VARIABLE_DEF
VARIABLE_DEF[124x4]	VARIABLE_DEF	124	4	VARIABLE_DEF
VARIABLE_DEF[127x4]	VARIABLE_DEF	127	4	VARIABLE_DEF
VARIABLE_DEF[130x4]	VARIABLE_DEF	130	4	VARIABLE_DEF
VARIABLE_DEF[133x4]	VARIABLE_DEF	133	4	VARIABLE_DEF
VARIABLE_DEF[138x4]	VARIABLE_DEF	138	4	VARIABLE_DEF
VARIABLE_DEF[141x4]	VARIABLE_DEF	141	4	VARIABLE_DEF
CTOR_DEF[147x4]	CTOR_DEF	147	4	CTOR_DEF
METHOD_DEF[165x4]	METHOD_DEF	165	4	METHOD_DEF
MODIFIERS[165x4]	MODIFIERS	165	4	MODIFIERS
TYPE[165x11]	TYPE	165	11	TYPE
setTabWidth[165x16]	IDENT	165	16	setTabWidth
([165x27]	LPAREN	165	27	(
PARAMETERS[165x28	PARAMETERS	165	28	PARAMETERS
)[165x41]	RPAREN	165	41	)
{[166x4]	SLIST	166	4	{
METHOD_DEF[171x4]	METHOD_DEF	171	4	METHOD_DEF

3. My contribution to the Checkstyle

At first, it was a bit hard for me to find a way to contribute to checkstyle. The best way was to use it in real conditions and that was the right moment as I had started working on a small project to which some friends intended to contribute too. So, I was trying to find one more function, which could also be used in the case of checking large projects to make the code more understandable for the intensive readers.

3.1 New check, ClassblanknessCheck.java

The first thing that came to my mind was code understandability. An important factor for that is the code's clarity. In many cases, although the code is not that difficult, code reading ends to be a difficult task. So, I found out that a check which would count the percentage of the blank lines of the block of a class would be really useful for these reasons. Having many blank lines among the code lines, makes a class easier to read. What I just had to do, was to create a class with the name of the check, create a setter method(*setMinBlankLines(int aMinBlankLines*)) which sets the minimum percentage of the block's blank lines and a *visitToken(DetailAST aAST)* which calls the method *blanknessCounter(String[] class_lines, int startLine, int finishLine),* and implements the check by reporting a message if the output of the *blanknessCounter* method is lower than *mMinBlankLines,* a private double variable. *VisitToken* method has this standard name, as every check extends *Check* abstract class and it is the method that manipulates the source's tree nodes. There is also an *AutomaticBean* class, which manipulates the setter methods by transforming them to properties, which can be set in the configuration (xml) file that calls the check and sets its properties' values.
Added code:
new class created, ***ClassblanknessCheck.java***
in com.puppycrawl.tools.checkstyle.checks.whitespace package.

3.2 Improvement of RedundantImportCheck.java functionality

After having a look in the project's tracker link, I found that ReduntantImportCheck could just test if there is a duplicated import, an

import of java.lang package or if an import comes from the same package as the current package. It could not find any duplication if the same package was imported twice from different hierarchy level. For example, duplication like:
import java.io.*;
import java.io.File;
could not be found and reported, something that is important, as the class File of the example is used twice. To implement this, one more check should be add in the VisitToken method to check every import, if it starts with the body of a wild-card import.
Added code:
*- An "if" statement was added to **RedundantImportCheck.java** to check if an import is already imported by a higher hierarchy level wild-card import.*

3.3 New properties of StrictDuplicateCodeCheck

Duplication code check, although working in a simple line by line way by converting each of the source lines to a checksum which is checked against to find duplicates, it ignores by default the import statements but includes javadoc and whitespace lines. This check can be used in many ways but it lacks flexibility. For this reason, adding some properties to this check, despite its easy implementation, would make it much more user configurable and useful.
Added code:
*- Two setter methods were added in **StrictDuplicateCodeCheck.java**, to set the values of the private boolean variables, **mIgnoreImports** and **mIgnoreComments**, which then determine if the imports and the javadoc comments will be ignored by the check or not. The variables' default values are left as they were.*
*- An "if" statement was added in the **calcChecksum(String aLine)** to check the values of the added variables, **mIgnoreImports** and **mIgnoreComments**, and finally calculate the checksum of every line (by calling **calcChecksum(String aLine)** of the super class) or ignore it by returning the **IGNORE** variable's value.*

3.4 A new reporting method, JavaLogger

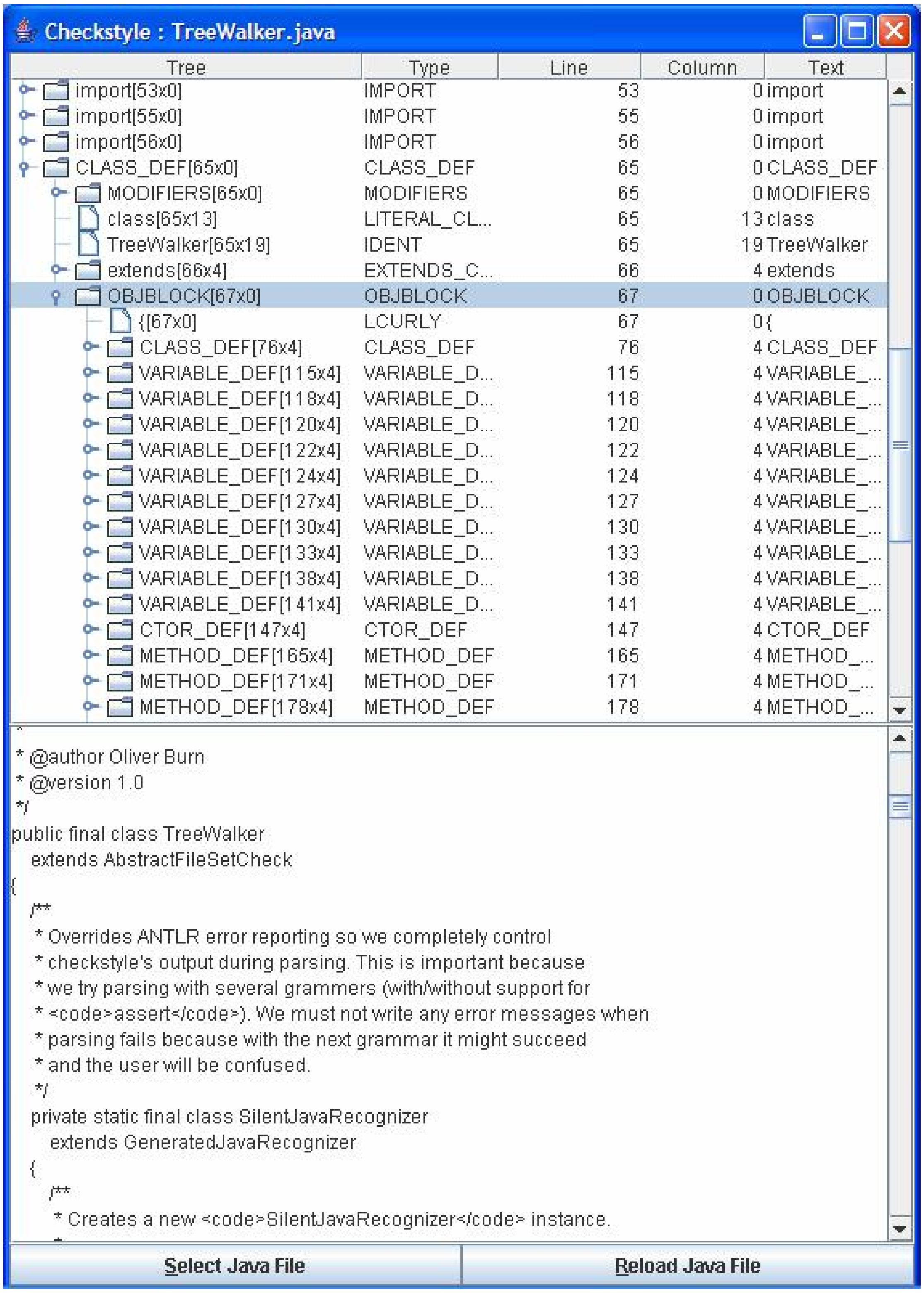
After talking with the course's Professor, I was given the idea of creating a new reporting method. The report messages can appear next to the corresponding line of the source code instead of reporting the file name, line number and the message, like *DefaultLogger* and *XMLLogger* do. All the source code of the checked java files is copied to the output file and the checkstyle messages next to the corresponding lines in the form of java comments.

Added code:
*- New file, **JavaLogger.java**, implements the new reporting way (which implements **AuditListener** interface). It keeps two ArrayLists, **mJavaFilesList** and **mErrorsMessageList,** which hold the names of the checked files and the error messages. While reading the lines of the checked files, it copies them to the output and adds the messages next to the file's line number that is mentioned in every error message.*
*- One more possible value to the project's format argument is added in **Main.java** of **com.puppycrawl.tools.checkstyle** package. When "-f java" is given when checkstyle is called from the command line, **JavaLogger.java** is used as an **AuditListener**.*

3.5 Changes to Checkstyle's GUI

Although left to the end of my contribution, I am really interested in making some more changes to checkstyle's GUI, as I did not have much time to do as many as I intended. May be I will be able to go on with some more changes in the future and this is something that I am going to ask from the project's administrator. What I managed to do was to create a text area, where the source code of the selected java file appears. Also, every time a file is selected, its name appears on the window's title. The changes I would like to go on with are to highlight the line of the source code when an AST node of the abstract tree is selected, or to open the corresponding AST node of the tree when a code's line is selected or execute individual checks and display the error messages on the text area I have already created.
Added code:
*- to **ParseTreeInfoPanel.java** in com.puppycrawl.tools.checkstyle.gui package. A command was added in **openFile(File aFile, final Component aParent)** to change the frame's title to contain the selected file's name. Also some code was added in the class's constructor which creates a JTextArea in the frame's panel. When a file is selected, the source code lines of that file are copied in the JTextArea, which was added in a JScrollPane, so that the user can scroll up and down to view the source of the selected file. The code that reads the file's source code and copies it to the JTextArea was added in **openFile(File aFile, final Component aParent),** which is called from the **FileSelectionAction class.***

Here the picture shows how the GUI looks after the changes I made.



Tree	Type	Line	Column	Text
import[53x0]	IMPORT	53	0	import
import[55x0]	IMPORT	55	0	import
import[56x0]	IMPORT	56	0	import
CLASS_DEF[65x0]	CLASS_DEF	65	0	CLASS_DEF
MODIFIERS[65x0]	MODIFIERS	65	0	MODIFIERS
class[65x13]	LITERAL_CL...	65	13	class
TreeWalker[65x19]	IDENT	65	19	TreeWalker
extends[66x4]	EXTENDS_C...	66	4	extends
OBJBLOCK[67x0]	OBJBLOCK	67	0	OBJBLOCK
{[67x0]	LCURLY	67	0	{
CLASS_DEF[76x4]	CLASS_DEF	76	4	CLASS_DEF
VARIABLE_DEF[115x4]	VARIABLE_D...	115	4	VARIABLE_...
VARIABLE_DEF[118x4]	VARIABLE_D...	118	4	VARIABLE_...
VARIABLE_DEF[120x4]	VARIABLE_D...	120	4	VARIABLE_...
VARIABLE_DEF[122x4]	VARIABLE_D...	122	4	VARIABLE_...
VARIABLE_DEF[124x4]	VARIABLE_D...	124	4	VARIABLE_...
VARIABLE_DEF[127x4]	VARIABLE_D...	127	4	VARIABLE_...
VARIABLE_DEF[130x4]	VARIABLE_D...	130	4	VARIABLE_...
VARIABLE_DEF[133x4]	VARIABLE_D...	133	4	VARIABLE_...
VARIABLE_DEF[138x4]	VARIABLE_D...	138	4	VARIABLE_...
VARIABLE_DEF[141x4]	VARIABLE_D...	141	4	VARIABLE_...
CTOR_DEF[147x4]	CTOR_DEF	147	4	CTOR_DEF
METHOD_DEF[165x4]	METHOD_DEF	165	4	METHOD_...
METHOD_DEF[171x4]	METHOD_DEF	171	4	METHOD_...
METHOD_DEF[178x4]	METHOD_DEF	178	4	METHOD_...

```
 * @author Oliver Burn
 * @version 1.0
 */
public final class TreeWalker
    extends AbstractFileSetCheck
{
    /**
     * Overrides ANTLR error reporting so we completely control
     * checkstyle's output during parsing. This is important because
     * we try parsing with several grammers (with/without support for
     * <code>assert</code>). We must not write any error messages when
     * parsing fails because with the next grammar it might succeed
     * and the user will be confused.
     */
    private static final class SilentJavaRecognizer
        extends GeneratedJavaRecognizer
    {
        /**
         * Creates a new <code>SilentJavaRecognizer</code> instance.
```

4. APPENDIX

4.1 Source code

ClassBlanknessCheck.java

```
////////////////////////////////////////////////////////////////////////////////
// checkstyle: Checks Java source code for adherence to a set of rules.
// Copyright (C) 2001-2005  Oliver Burn
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
////////////////////////////////////////////////////////////////////////////////

package com.puppycrawl.tools.checkstyle.checks.whitespace;

import java.util.regex.*;

import com.puppycrawl.tools.checkstyle.api.Check;
import com.puppycrawl.tools.checkstyle.api.DetailAST;
import com.puppycrawl.tools.checkstyle.api.FullIdent;
import com.puppycrawl.tools.checkstyle.api.TokenTypes;


/**
 * <p>
 * Checks for the percentage of the blank lines of
 * a class's block. As blank lines are considered:
 * </p>
 *<ul>
 *  <li>These which are totally blank and match with regexp="^\s*$"</li>
 *  <li>These which contain a '{' or '}' character with whitespace characters
around and
 *  match with regexp="^\s*(}|\{)\s*$"</li>
 *</ul>
 * <p>
 * An example of how to configure the check is:
 * </p>
 * <pre>
 * &lt;module name="ClassBlankness"&gt;
 *   &lt;property name="minBlankLines" value="int"/&gt;
 * &lt;/module&gt;
 * </pre>
 *
 * Compatible with Java 1.5 source.
 *
 * @author Ioannis Sermetziadis
 * @version 1.0
 */
public class ClassBlanknessCheck
       extends Check {

   /** default value for the minimum blank lines in a class*/
   private static final int MIN_BLANK_LLINES_IN_A_HUNDRENT_OF_LINES
= 20;

   /** give default value to mMinBlankLines*/
   private int mMinBlankLines =
MIN_BLANK_LLINES_IN_A_HUNDRENT_OF_LINES;

   /** variables to keep the lines of the first and last chind of a class's block */
   private static int mCheckStartLine, mCheckStartColumn, mCheckFinishLine;

   /** Creates a new instance of ClassBlankness */
   public ClassBlanknessCheck() {
   }


   /** {@inheritDoc} */
   public int[] getDefaultTokens()
   {
      return new int[]
      {TokenTypes.CLASS_DEF,
      };
   }
   public void setMinBlankLines(int aMinBlankLines)
   {

      mMinBlankLines = aMinBlankLines;
   }

   /** {@inheritDoc} */
   public void visitToken(DetailAST aAST)
   {

      if (aAST.getType() == TokenTypes.CLASS_DEF) {

         DetailAST objBlock = aAST.getLastChild();
         DetailAST child = objBlock.getLastChild();

         while(true) {
        //the class's block starts from the first '{'
            if(child.getType() == TokenTypes.LCURLY) {
                mCheckStartLine = child.getLineNo();
                mCheckStartColumn = child.getColumnNo();
                break;
            }
            //the class's block ends with the last '}'
            else if(child.getType() == TokenTypes.RCURLY) {

                mCheckFinishLine = child.getLineNo();
            }

            child = child.getPreviousSibling();

          }


        String[] lines = getLines();
        int blankness = blanknessCounter(lines, mCheckStartLine,
mCheckFinishLine);
          if(blankness < mMinBlankLines) {

          log(mCheckStartLine, mCheckStartColumn, "blank.not.enough");
          }
        }

    }

   /** Counts the blankness of a block of a class and gives about an integer
    * with the percentage of a class's block's lines that are blank.*/

    private int blanknessCounter(String[] class_lines, int startLine, int finishLine)
    {
        int blank_counter = 0;
        int block_length;
        Pattern regExpPatern = Pattern.compile("^\\s*(}|\\{)?\\s*$");
        for(block_length = startLine - 1; block_length < finishLine; block_length++) {

          Matcher regExpMatcher =
regExpPatern.matcher(class_lines[block_length]);

          if(regExpMatcher.matches())
             blank_counter++;

        }
        return (int)(((double)blank_counter / (double)(finishLine - startLine)) * 100);
    }
}
```

Part of RedundantImportCheck.java, where I added code

```
while (it.hasNext()) {
            final FullIdent full = (FullIdent) it.next();
            final String stringToCheck;

            //this algorithm can be added in api.Utils if needed several times
            if(full.getText().indexOf(".*") != -1)
              stringToCheck = full.getText().substring(0,
full.getText().lastIndexOf('.'));
            else
              stringToCheck = full.getText();

            if ((imp.getText().startsWith(stringToCheck) &&
!stringToCheck.equals(full.getText())) ||
                  (imp.getText().equals(stringToCheck)) &&
stringToCheck.equals(full.getText())) {
               log(aAST.getLineNo(),
                  aAST.getColumnNo(),
                  "import.duplicate",
                  new Integer(full.getLineNo()),
                  imp.getText());
            }
         }
```

Code I added in StrictDuplicateCodeCheck.java

```
/**
    * Sets if imports will be ignored or not before the check starts
    *
    *@param mIgnoreImports must be set before
    * triggering a 'duplicate code' message. Default value = true
    */
   public void setIgnoreImports(boolean aIgnoreImports)
   {
     mIgnoreImports = aIgnoreImports;
   }

   /**
    * Sets if comments will be ignored or not before the check starts
    *
    *@param mIgnoreComments must be set before
    * triggering a 'duplicate code' message. Default value = false
    */
   public void setIgnoreComments(boolean aIgnoreComments)
   {
     mIgnoreComments = aIgnoreComments;
   }
```

JavaLogger.java

```
////////////////////////////////////////////////////////////////////////////////
// checkstyle: Checks Java source code for adherence to a set of rules.
// Copyright (C) 2001-2005  Oliver Burn
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
////////////////////////////////////////////////////////////////////////////////
package com.puppycrawl.tools.checkstyle;

import java.io.OutputStream;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.ArrayList;

import com.puppycrawl.tools.checkstyle.api.AuditEvent;
import com.puppycrawl.tools.checkstyle.api.AuditListener;
import com.puppycrawl.tools.checkstyle.api.AutomaticBean;
import com.puppycrawl.tools.checkstyle.api.SeverityLevel;
import com.puppycrawl.tools.checkstyle.api.Utils;


/**
 * This class implements a Java logger.
 * Gives as an output a java file that contais the code
 * of the source file and the error logs as commnents next
 * to the corresponding lines of the code.
 *
 * @author Ioannis Sermetziadis
 */
public class JavaLogger
    extends AutomaticBean
    implements AuditListener
{
    /** cushion for avoiding StringBuffer.expandCapacity */
    private static final int BUFFER_CUSHION = 12;

    /** where to print the error messages **/
    private PrintWriter mOutputWriter;

    /** close error stream after use */
```

```
private boolean mCloseOutput;

/** keeps errors's info and message */
private ArrayList mErrorsMessageList;

/** keeps a List of the files already checked and have an error */
private ArrayList mJavaFilesList;


/** Creates a new instance of JavaLogger */
public JavaLogger(OutputStream aOS, boolean aCloseStreamsAfterUse)
{
    mOutputWriter = new PrintWriter(aOS);
    mCloseOutput = aCloseStreamsAfterUse;
    mJavaFilesList = new ArrayList();
    mErrorsMessageList = new ArrayList();

}

public void addError(AuditEvent aEvt)
{
  final SeverityLevel severityLevel = aEvt.getSeverityLevel();
  if (!SeverityLevel.IGNORE.equals(severityLevel)) {

    final String fileName = aEvt.getFileName();
    final String message = aEvt.getMessage();


    if(!mJavaFilesList.isEmpty()) {
        if(!mJavaFilesList.contains(fileName)) {
            mJavaFilesList.add(fileName);

        }
    }
    else {
        mJavaFilesList.add(fileName);
    }

    // avoid StringBuffer.expandCapacity
    final int bufLen = fileName.length() + message.length()
      + BUFFER_CUSHION;
    final StringBuffer sb = new StringBuffer(bufLen);

    sb.append(fileName);
    sb.append(':').append(aEvt.getLine());
    if (aEvt.getColumn() > 0) {

      sb.append(':').append(aEvt.getColumn());
    }
    if (SeverityLevel.WARNING.equals(severityLevel)) {

       sb.append(": warning");
    }
    sb.append(": ").append(message);
    mErrorsMessageList.add(sb.toString());

    }

}

/** {@inheritDoc} */
public void addException(AuditEvent aEvt, Throwable aThrowable)
{
   mErrorsMessageList.add("Error auditing: " + aEvt.getFileName());

}

/** {@inheritDoc} */
public void auditStarted(AuditEvent aEvt)
{
    mOutputWriter.println("// Starting audit...");
}

/** {@inheritDoc} */
public void fileFinished(AuditEvent aEvt)
{
}

/** {@inheritDoc} */
public void fileStarted(AuditEvent aEvt)
{
}

/** {@inheritDoc} */
public void auditFinished(AuditEvent aEvt)
{
    closeStreams();
    mOutputWriter.println("Audit done.");
    if (mCloseOutput) {
      mOutputWriter.close();
    }
}

/**
 * Collects and flushes the output streams and closes them if needed.
 */
protected void closeStreams()
{   StringBuffer sb = new StringBuffer();

try
{
    for(int i = 0; i < mJavaFilesList.size(); i++) {
       String fileName = mJavaFilesList.get(i).toString();
       String[] contentLines = Utils.getLines(fileName);

          for(int u = 0; u < contentLines.length; u++) {
             sb.append(contentLines[u]);

             for(int y = 0; y < mErrorsMessageList.size(); y++) {
                if((mErrorsMessageList.get(y).toString().indexOf(fileName) != -1)) {
                    if(mErrorsMessageList.get(y).toString().startsWith("Error")) {
                        sb.append("//" + mErrorsMessageList.get(y).toString());
                        mErrorsMessageList.remove(y);
                    }
                    else  if(mErrorsMessageList.get(y).toString().indexOf(fileName  +
":" + (u + 1)  + ":") != -1) {
                        sb.append("//" + mErrorsMessageList.get(y).toString());
                        mErrorsMessageList.remove(y);
                    }
                }

              }
            sb.append("\n");

            }
        }
    }
    catch(IOException e)
    {
      System.out.println("Cannot access the files, giving up: "
                  + e.getMessage());
    }
    mOutputWriter.println(sb);
    mOutputWriter.flush();
    }

    public ArrayList getErrors() {
      return mErrorsMessageList;
    }
}
```

Part of Main.java, where I added code

```
private static AuditListener createListener(CommandLine aLine,
                                            OutputStream aOut,
                                            boolean aCloseOut)
    {
      final String format =
          aLine.hasOption("f") ? aLine.getOptionValue("f") : "plain";

      AuditListener listener = null;
      if ("xml".equals(format)) {
          listener = new XMLLogger(aOut, aCloseOut);
      }
      else if ("plain".equals(format)) {
          listener = new DefaultLogger(aOut, aCloseOut);
      }
      else if ("java".equals(format)) {
          listener = new JavaLogger(aOut, aCloseOut);
      }
      else {
          System.out.println("Invalid format: (" + format
                      + "). Must be 'plain', 'xml' or 'java'.");
```

```
        usage();
      }
      return listener;    }
```

Part of ParseTreeInfoPanel.java, where I added code

```
public ParseTreeInfoPanel()
   {
      setLayout(new BorderLayout());

      DetailAST treeRoot = null;
      mParseTreeModel = new ParseTreeModel(treeRoot);
      mTreeTable = new JTreeTable(mParseTreeModel);
      final JScrollPane sp = new JScrollPane(mTreeTable);
      this.add(sp, BorderLayout.NORTH);

      final JButton fileSelectionButton =
         new JButton(new FileSelectionAction());

      reloadAction = new ReloadAction();
      reloadAction.setEnabled(false);
      final JButton reloadButton = new JButton(reloadAction);

      mJTextArea = new JTextArea(20, 15);
      mJTextArea.setEditable(false);

      final JScrollPane sp2 = new JScrollPane(mJTextArea);
      this.add(sp2, BorderLayout.CENTER);

      final JPanel p = new JPanel(new GridLayout(1,2));
      this.add(p, BorderLayout.SOUTH);
      p.add(fileSelectionButton);
      p.add(reloadButton);

      try {
         // TODO: creating an object for the side effect of the constructor
         // and then ignoring the object looks strange.
         new FileDrop(sp, new FileDropListener(sp));
      }
      catch (TooManyListenersException ex)
      {
        showErrorDialog(null, "Cannot initialize Drag and Drop support");
      }

   }

public void openFile(File aFile, final Component aParent)
   {
      if (aFile != null) {
         try {
            Main.frame.setTitle("Checkstyle : " + aFile.getName());
            DetailAST parseTree = parseFile(aFile.getAbsolutePath());
            mParseTreeModel.setParseTree(parseTree);
            mCurrentFile = aFile;
            mLastDirectory = aFile.getParentFile();
            reloadAction.setEnabled(true);

            String[] sourceLines = Utils.getLines(aFile.getName());
            /* clean the text area before inserting the lines of the new file **/
            if(mJTextArea.getText().length() != 0)
               mJTextArea.replaceRange("", 0, mJTextArea.getText().length());

            /* insert the contents of the file to the text area **/
            for(int i = 0; i < sourceLines.length; i++) {
               mJTextArea.append(sourceLines[i] + "\n");

            }
         /** move back to the top of the file */
         mJTextArea.moveCaretPosition(0);

         }
         catch (IOException ex) {
            showErrorDialog(
                  aParent,
                  "Could not open " + aFile + ": " + ex.getMessage());
         }
         catch (ANTLRException ex) {
            showErrorDialog(
                  aParent,
                  "Could not parse " + aFile + ": " + ex.getMessage());
         }
      }
   }
```

4.2 Contact with the project’s administrator

<u>I wrote:</u>

Hi, my name is John Sermetziadis and I am an undergraduate student of Software Engineering in Greece. As far as a course is concerned, “Advance Topics in Software Engineering” I have to make a contribution to an open-source project. So, I decided that I would like to contribute to “checkstyle".

Do you have any suggestions of any new kind of checks, I could develop or of any changes that you would like to be made to the existing checks?

Some ideas I have, as far as I am using the software, are:
1. There could be a check of code “clearness” which would simply check how many blank lines (as a parameter) there should be inside a 100 lines class or how many spaces there should be inside an 10,000 char class so that the code to be at a minimum level as clear as possible to be read by other programmers.

2. I noticed that when using the RedundantImport it doesn’t report it if I import twice a class like that:

```
import java.io.*;
import java.io.File;
```

Don’t you think it should? The class File isn’t imported twice?

3. Do you think it’s useful to extend the “UnusedImports” so that I can support wild-card imports like import java.io.*; ?

4. Finally, in my opinion, in the StrictDuplicateCode check there should be provided some properties’ options (boolean properties), whether to ignore or not javadoc comments or whitespace lines between methods. What do you think about that?

Thank you for your time! If you have any ideas, I would be glad to hear them.
--
This message has been sent to you, a registered SourceForge.net user, by another site user, through the SourceForge.net site. This message has been delivered to your SourceForge.net mail alias. You may reply to this message using the "Reply" feature of your email client, or using the messaging facility of SourceForge.net at: https://sourceforge.net/sendmessage.php?touser=1473526

Oliver Burn wrote:

Hi John,

Are you based in Athens? I fondly remember visiting Greece in 1993 while backpacking in Europe. Had a great time there.

Glad you would like to contribute to Checkstyle. Interesting to hear that it is a requirement for part of your course. We are always keen to get contributions, and have written a page outlining the best way – it is at http://checkstyle.sourceforge.net/contributing.html.

For ideas of what Checks people are interested in, look at the Requests area at http://sourceforge.net/tracker/?atid=397081&group_id=29721&func=browse.
There are also ideas mixed in with the bugs at
http://sourceforge.net/tracker/?atid=397078&group_id=29721&func=browse.

To answer you specific questions:

1. It is an interesting idea. Curious as to what you think is a good measure. In a 100 line class, how many blanks lines should I have?

2. Technically you are correct, and this would be a good feature. Personally I never use the star import format so it has not been a problem for me.

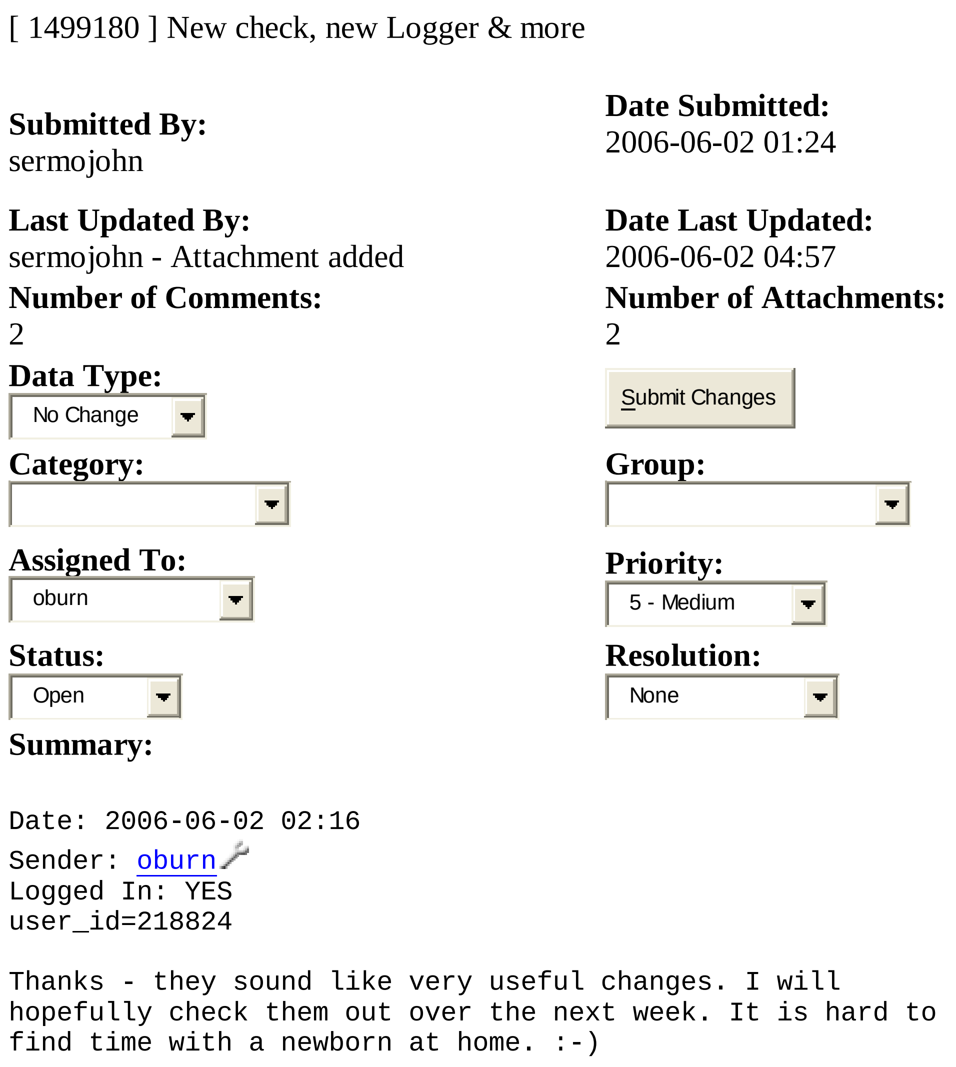
3. This would be a hard change. If you look at the current implementation, it uses a very simple algorithm (actually a brain dead one), that in practice is very accurate. The only downside is that it cannot handle star imports. To support star imports would introduce the need for the check to understand Java types.

4. A good idea.

Let me know how you go.

Regards,
Oliver

4.3 Track upload and administrator's comment



[1499180] New check, new Logger & more

Submitted By: sermojohn	**Date Submitted:** 2006-06-02 01:24
Last Updated By: sermojohn - Attachment added	**Date Last Updated:** 2006-06-02 04:57
Number of Comments: 2	**Number of Attachments:** 2
Data Type: No Change	Submit Changes
Category:	**Group:**
Assigned To: oburn	**Priority:** 5 - Medium
Status: Open	**Resolution:** None

Summary:

```
Date: 2006-06-02 02:16
Sender: oburn
Logged In: YES
user_id=218824

Thanks - they sound like very useful changes. I will
hopefully check them out over the next week. It is hard to
find time with a newborn at home. :-)
```