U. S. ELECTION ASSISTANCE COMMISSION
VOTING SYSTEM TESTING AND CERTIFICATION PROGRAM
1201 New York Avenue, NW, Suite 300
Washington, DC. 20005

To: Registered Manufacturers and Voting System Test Laboratories

From: Brian Hancock, Director

Date: March 21, 2012

Re: Automated Source Code Analysis Tools

Over the past year, the EAC received a number of questions regarding the use of automated tools to satisfy the source code review requirements of the 2005 VVSG. As you know, the current manual source code review methods are expensive and consume an estimated 30-40% of a testing campaign's total time. Manual review may lead to inconsistent application of the source code review requirements of the 2005 VVSG.

The source code requirements of the 2005 VVSG can be divided into the following categories:

- Software integrity
- Software modularity and programming
- Control constructs
- Naming conventions
- Commenting conventions

Three categories of source code related 2005 VVSG requirements can be fully or partially automated: software integrity, software modularity and programming, and control constructs. Determining conformance to the naming convention and commenting conventions requirements cannot generically be performed by automated tools, as the 2005 VVSG does not specify a particular naming or commenting convention for manufacturers to follow. Human analysis or tool customization is required to verify conformance.

RFI 2010-02 clarifies the binary option of choosing either the coding convention specified by the 2005 VVSG or an industry-accepted coding standard for a specific language and voting system component. Used in the correct context, automated tools can assist in determining conformance with the 2005 VVSG's coding convention or an industry-accepted coding standard. Tools for assessing conformance to an industry-accepted coding standard already exist, are widely used outside the voting system industry, and are ready to be leveraged. Conversely, tools for assessing conformance to the 2005 VVSG's coding convention will likely need to be developed.

In an effort to test this concept, last year the EAC suggested Wyle Laboratories pilot the use of automated tools before endorsing their use within the Program (letter attached). Wyle piloted this concept and produced a Test Report for the EAC (Test Report attached). Wyle reported positive results in assessing adherence to the 2005 VVSG using automated tools and reported drastic cost and time savings. It is important to remember that purely automated testing of code is not advisable nor is it the goal of this effort. Both manual and automated review should be used in the correct context.

Effective immediately, the EAC supports the use of automated tools whenever possible to assess conformance to the coding convention specified with the 2005 VVSG and other industry-accepted coding standards. Additionally, the EAC encourages manufacturers to use these tools before entering the EAC's Certification Program to lower the time and expense associated with voting system testing and certification. A specific tool or methodology will not be dictated, nor will the EAC validate the correct operation of a tool. VSTLs must determine if a tool functions as intended. To assist in determining conformance to the naming conventions and commenting conventions categories of requirements using manual source code review methods, the EAC will allow VSTLs to sample the submitted source code. The results of the sample review and the VSTLs engineering analysis will be provided to the EAC for review, before the EAC makes a determination on acceptance.

Please let me know if you have any additional questions.

Sincerely,

Brian J. Hancock

Brian J. Hancock, Director
Voting System Testing and Certification

Attached:

- *Use of Automated Source Code Review Tools Report (Pilot)*
- *The use of automated source code review tools*

U. S. ELECTION ASSISTANCE COMMISSION
VOTING SYSTEM TESTING AND CERTIFICATION PROGRAM
1201 New York Avenue, NW, Suite 300
Washington, DC. 20005

August 26, 2011

Frank Padilla
Wyle Laboratories
7800 Highway 20 West
Huntsville, Al 35806

Subject: The use of automated source code review tools

Dear Mr. Padilla,

Recently Wyle Laboratories approached the Election Assistance Commission Testing and Certification Division about the use automated source code review tools. We understand that Wyle would like to begin using automated source code review tools as an alternative to full manual line by line examination of source code. The EAC is committed to lowering the cost and reducing the time needed to complete certification testing. The use of automated tools may help to achieve both of those goals; however, the EAC would like to proceed cautiously. The EAC suggests that Wyle initiate a pilot run of these tools before instituting automated source code review on a wider basis. The conditions of the pilot are:

- The pilot can be performed on source code currently in our program, entering our program, or outside of our program.
- The pilot must focus languages with a well established or frequently used set of coding standards, (e.g., Java, C++).
- Wyle must submit a report on the results of the pilot. The report must include:
    - Accuracy of the tool;
    - Comparison of resources used with automated vs. manual review, (i.e. time and materials);
    - Ability to test conformity to the 2005 VVSG and other conventions;
    - Sampling methods used and percentage of code sampled for comment quality.

If you have additional questions, please do not hesitate to contact me.

Sincerely,

[illegible]

Director, Testing and Certification

wyle

7800 Highway 20 West
Huntsville, Alabama 35806
Phone (256) 837-4411
Fax (256) 721-0144
www.wyle.com

USE OF AUTOMATED SOURCE CODE REVIEW TOOLS REPORT (PILOT)

Prepared by:

[illegible] 10/20/11

Jim Falwell, Source Code Project Engineer

NVLAP LAB CODE 200771-0

U.S. Election Assistance Commission
VSTL
EAC Lab Code 0704

TABLE OF CONTENTS

13 pages total, including cover page and TOC

1.0 INTRODUCTION

This review covers an automated source code review pilot system submitted for evaluation. The pilot involved the automated evaluation of a submitted set of source code as to its adherence to a published industry-accepted standards document. The automated tool was evaluated for its ability to identify and verify source code non-compliance to standards in two areas. First the industry-accepted standard for the language submitted and second any ability the tool might contain to verify compliance to the EAC VVSG 2005.

The pilot evaluation will be based on two documents.

1. EAC Request for Interpretation (RFI) 2010-02, Located at: http://www.eac.gov
2. Souce code pilot letter_8-26-11-Final (4).pdf [Emailed from Brian Hancock on August 26th, 2011]

2.0 ACCURACY OF AUTOMATED TOOL

The automated tool evaluated in this pilot test report consisted of the following:

1. NetBeans – Version 6.9.1
2. CheckStyle – Version 5.3
3. Customized checkstyle xml file (file may contain standard and/or customized checks)
4. Voting System Software - Java code.
5. Java Code Conventions:

   http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

   (NOTE: Link provided by vendor and downloaded independently by Wyle.)

The Java code conventions identified in Step 4, above, were evaluated for coding convention requirements. Based on this published version of Oracle/Sun Java Coding Conventions, most of the Java Coding Conventions are suggestions and not requirements. However, there are some conventions that use absolute language so as to imply requirements. There are a few non-absolute requirements that use the phraseology "should always" or "should never". These instances were interpreted as "shall" and "shall not" and were interpreted as requirements.

There were nine sections (with 14 examples to be validated) identified in the Java Code Conventions September 12, 1997 document that contained requirements. There was one section with the "should always" type of wording discussed above.

The coding convention requirements found in the published standard are documented in Appendix A of this report "Java Coding Convention Requirements of Java Code Conventions September 12, 1997" hereinafter referred to as Appendix A.

3.0 COMPARISON OF AUTOMATED PROCESS VS HISTORICAL MANUAL REVIEW

There are over 513,262 lines of code in the application used for this test report. For the traditional method of manual review, 700 lines of code reviewed per hour is a reasonable production estimate based on historical review data (for the initial pass of source code review). There are usually multiple re-reviews required depending upon the amount of discrepancies found. This method would take 733.23 man hours to perform the initial code review for this application [NOTE: Historically, Wyle VSTL has reviewed code in the range of 1 to 200 re-reviews for a given application. Multiple factors contribute to the number of necessary re-reviews. These factors include size of application, number of discrepancies, level of initial conformance to the EAC 2005 VVSG standards, vendor response and attention to standards detail, among other things.]

For the automated approach, the review process from start to finish would take approximately 100 man hours (for the initial pass of source code review). The amount of re-reviews or full reviews would depend on the audit of the code and amount of discrepancies found.

This pilot evaluation time savings between the historical fully manual review and the automated source code review processes are in the vicinity of 600 man hours. This significant time savings speaks volumes as to which method is preferable based on both cost and time to completion.

The comparison results are summarized in Table 3.1, below.

TABLE 3.1 Automated Process vs Historical Manual Review

Review Method	LOC (Lines Of Code)	Review Rate	Review Man Hours	Total Review Man Hours
Manual LOC Tested (Historical Manual Review)	513,262	700 lines of code reviewed per hour	733.23 man hours (Initial Full Review)	733 total man hours for a Manual Code Review process to be completed.
Automated LOC Tested. (Part of the Automated Review Process)	513,262	5132 per hour (NOTE: Includes automated system setup, Java Coding Conventions and Checkstyle verification.)	~24 man hours (Initial Full Review)	100 total man hours for an Automated Code Review process to be completed.
11% Manual LOC Tested Under Automated Pilot (Part of the Automated Review Process)	~57,112 *	750 lines of code reviewed per hour (NOTE: Higher LOC reviewed per hour when only reading comments/headers.)	~76 man hours	

* NOTE: This data is discussed in Section 6.0 of this report.

4.0 TOOL ABILITY TO TEST CONFORMITY TO PUBLISHED JAVA CODING CONVENTIONS

The tool's ability to test for compliance to the Java Coding Conventions relies primarily upon Checkstyle. Checkstyle is an automation tool that handles different languages with built-in and customizable checks (coding convention verifications). For the Java language, Checkstyle has over 140 built-in language checks and can be customized to handle other checks as needed.

Each Java Coding Convention requirement discovered in the published document chosen by the manufacturer was evaluated for flagging a non-compliance or not. This testing was accomplished by creating test code that forced each issue in a modified Java file contained in the source code submitted for evaluation. The test results of the requirements or pseudo requirements evaluated follows are detailed below. [Note that the term, "disallowed", means that the check in question worked correctly and disallowed the non-compliance.]

Section 6.1 Number Per Line (two examples to check for):

1) Declaring variables and functions on the same line.
   This was flagged by the automated tool as expected - disallowed.
2) Declaring different types on the same line.
   This was flagged by the automated tool as expected - disallowed.

Section 6.2 Placement (two examples to check for)

1) First use declaration of variables down in blocks of code.
   Not flagged as expected – this was allowed.
2) Local Declarations that hide declarations at higher levels.
   This was flagged by the automated tool as expected - disallowed.

Section 7.1 Simple Statements

1) Using the comma operator to group multiple statements.
   N/A – NetBeans/Java tool setup disallows.

Section 7.2 Compound Statements

1) Braces are used around all statements.
   This was flagged by the automated tool as expected - disallowed.

Section 7.8 Switch Statements

1) Disallow fall through on "case" statements
   This was flagged by the automated tool as expected - disallowed.
2) Intentional "case" fall through – add a comment of: /* falls through */
   This check was flagged by the automated tool as expected - disallowed.
   (NOTE: Comment worked as stated.)

Section 8.1 Blank Lines

1) This is a "should always" section. It was not enforced by Checkstyle, but it is not a requirement. N/A

4.0 TOOL ABILITY TO TEST CONFORMITY TO PUBLISHED JAVA CODING CONVENTIONS (CONTINUED)

Section 8.2 Blank Spaces

1) No spaces in assignments such as: a+=c+d;
   Not flagged as expected – this was allowed.
2) No spaces around unary operators, such as a++; // a ++;
   This was flagged by the automated tool as expected - disallowed.

Section 9 Naming Conventions

1) Except for variables, all instance, class and class constants are in mixed case with a lowercase first letter. Internal words start with a capital letter.
   This was flagged by the automated tool as expected - disallowed.

Section 10.4 Variable Assignments

1) Don't use assignment operator where easily confused with equality operator, such as: if(c++ = d++) {
   N/A – Java disallows
2) Should be written as example of "1" above in documentation: if((c++ = d++) != 0) {
   N/A – Java disallows
3) Similar example to "2" above using equality operator: if((c++ == d++) != true) {
   This was flagged by the automated tool as expected - disallowed.
   (NOTE: Either "Inner assignments should be avoided" or "Expression can be simplified" flags were indicated by automatic tool.)
4) Embedded assignments such as: d = (a = b + c) + r;
   This was flagged by the automated tool as expected - disallowed.

Section 10.5.4 Special Comments

1) N/A to automated tool. The verification of this requirement was deemed to require human observation.

Overall Checkstyle appeared to do a very thorough job of identifying the standard Java Coding Conventions within the automated tool. However, there were some apparent inconsistencies observed. For instance, Section 8.2 was enforced in its 8.2.2 rendering (unary operators restricted from having blank spaces separating the variable from the operator), but not in 8.2.1 (blank spaces not separating variables from some other types of operators such as '+=' or '+'). This appears to be an inconsistency in the implementation of blank space restriction within the Checkstyle tool. However, it is possible that a separate switch controls this difference in operator checking for surrounding blank spaces. All sections were found to be enforced except for sections 6.2.1 and 8.2.1. Some other sections were deemed not applicable.

Per EAC RFI 2010-02, the submitted code is compliant with the chosen published standard of Java Coding Conventions. Based on this RFI, it is not necessary for the submitted code to conform to any of the EAC 2005 VVSG Source Code standards other than Section 5.2.7a when a vendor elects to conform their code to a published industry-accepted standard (EAC 2005 VVSG – Vol I – Section 5.2.6a). Section 5.2.7a requirements were tested in this pilot by evaluating 10% of the code base manually – direct human line-by-line review – for compliance to section 5.2.7a. This 10% test process is detailed in section 6.0 Sampling Methods Used of this report.

5.0 TOOL ABILITY TO TEST CONFORMITY TO THE EAC VVSG 2005

At the request of the EAC's Source Code Pilot letter dated August 26, 2011, Wyle did a thorough analysis of the automated tool's ability to check for the EAC 2005 VVSG source code standards, even though the tool is not required to check for them per RFI 2010-02.

Wyle's analysis was also directed to the EAC 2005 VVSG (Volumes I and II) Section 5 standards for source code. These requirements were evaluated in terms of the Java language and any existing implementation in the standard or custom configuration of Checkstyle submitted by the Vendor. Where applicable in the Java language, test code was inserted into an original source file for each standard in both Volumes I and II in an attempt to evaluate whether a violation of that standard is flagged for non-compliance or not.

Wyle's analysis determined that many of the EAC 2005 VVSG requirements are not applicable to the Java programming language. For instance, Java does not have pointers, does not allow unbounded arrays (without throwing an exception), does not have macros, etc. Another observation was that many other of the EAC 2005 VVSG requirements would require human intervention and would either not be practical or possible to implement in automatic compliance-checking code. Examples of this include virtually all header and/or comment standards. Some requirements were not evaluated at all since they are rarely observed in voting system code, eg, Mixed-mode Operations, Separate and Consistent formats for status versus error messages, etc. There were several EAC 2005 VVSG requirements that the submitted system implemented and these were evaluated for compliance.

Wyle's testing involved writing code within a Java source file that would force each of the Java applicable EAC 2005 VVSG standards and then re-running the tool. The results of the analysis, along with the requirements that were waived (per RFI 2010-02) are presented below:

<u>**EAC VVSG 2005 – Volume I**</u>

5.2.3 Software Modularity and Programming

- VVSG - Vol I - 5.2.3b. – Verified forced test code as non-VVSG compliant.
- VVSG - Vol I - 5.2.3e. – Verified forced test code as non-VVSG compliant.

5.2.5 Naming Conventions

- VVSG - Vol I - 5.2.5c. – Verified forced test code as non-VVSG compliant.
- VVSG - Vol I - 5.2.5d. – Verified forced test code as non-VVSG compliant.

5.2.7 Comment Conventions

- VVSG - Vol I - 5.2.7b. – Verified forced test code as non-VVSG compliant.

<u>**EAC VVSG 2005 – Volume II**</u>

5.4.2 Assessment of Coding Conventions

- VVSG - Vol II - 5.4.2a – Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.
- VVSG - Vol II - 5.4.2b – Verified forced test code as non-VVSG compliant.
- VVSG - Vol II - 5.4.2f – Verified forced test code as non-VVSG compliant.

5.0 TOOL ABILITY TO TEST CONFORMITY TO THE EAC VVSG 2005 (CONTINUED)

- VVSG - Vol II - 5.4.2h - Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.
- VVSG - Vol II - 5.4.2i - Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.
- VVSG - Vol II - 5.4.2k – Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.
- VVSG - Vol II - 5.4.2l – Verified forced test code as non-VVSG compliant.
- VVSG - Vol II - 5.4.2m – Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.
- VVSG - Vol II - 5.4.2r – Verified forced test code as non-VVSG compliant.
- VVSG - Vol II - 5.4.2s – Verified forced test code as non-VVSG compliant.
- VVSG - Vol II - 5.4.2t – Verified some forced test code as non-VVSG compliant and did not verify other forced test code as non-VVSG compliant - Waived per RFI 2010-02 – Sporadic. Catches some standards violations and seems to miss others.
- VVSG - Vol II - 5.4.2u – Verified forced test code as non-VVSG compliant.
- VVSG - Vol II - 5.4.2v - Did not verify test code as non-VVSG compliant - Waived per RFI 2010-02.

6.0 SAMPLING METHODS USED

It was determined via discussion between the EAC and the VSTL that it would not be possible to automatically verify the quality of comments in either the code module headers or in-line comments within a given set of code. Evaluating quality of comments would require testing using human observation. Therefore, a percentage of the code was sampled to assess conformance without manually reviewing all the code. A percentage of not less than 3% and not more than 20% was discussed as potential percentage values. For this pilot test 10% of the total code base was chosen at random for the review of the quality of commenting contained therein.

The following analysis was used to determine the 10% lines of code (LOC) to be reviewed:

1. The number of lines of code for the entire application was determined. Based on exploratory and experienced based testing techniques, it had been duly noted in the past that sometimes entire directories of files might have one or more compliance violations for every file in a given directory. [NOTE: Experience-based test techniques can include exploratory testing, error guessing (anticipating defects based on experience), available defect data from other projects, and common knowledge about why software fails (in this case – where Wyle has noted non-compliance issues with comments in previous election software applications).] For example, an entire folder of files might have no comments in the code or every file might be missing headers.
2. Based on these test observation(s) detailed in Step 1, an attempt was made to select at least a single file from every directory present in the application to try to make sure we have a valid representative code subset for our analysis. [This step can potentially be only partially fulfilled in the event that a single file from every folder exceeds the 10% LOC target.]

6.0 SAMPLING METHODS USED (CONTINUED)

3. The lines of code for the selected code subset was then determined. If the code subset contained >= 10% of the application total LOC, then we stopped our analysis and used this subset of the code for review. If it was < 10% LOC, then we chose additional files from each folder at random until we obtained the needed 10% target.

For Example – using the submitted source code base:

Step 1: 513,262 lines of code. Therefore, 10% ~ 51,326.

Step 2: 161 files selected – at least one per application directory.

Step 3: 57,112 lines of code contained in step 2 code subset.

(Approximately 11% of the application code total.)

7.0 CONCLUSION

As stated in Section 6.0 of this report, 10% of the total LOC was chosen as the percentage of LOC to review manually for comment/header quality and compliance to the VVSG 2005 Section 5.2.7. This percentage together with the other criteria used, such as, at least a single file from every folder, worked well for this application's source code evaluation. However, the choosing of 10% is subjective in nature. It is feasible that for other applications source code, a lesser or greater percentage might be chosen based on the quality of commenting observed. For instance if every comment in the first 5% of the LOC evaluated was absolutely compliant and expressed a superior quality as to its communication and maintainability in content, then 5% of the total LOC might be deemed to be a sufficient manual review amount depending on other application factors such as the volume of directories within the application, etc. In other cases where a large number of non-compliant commenting is encountered, it might be deemed necessary to increase the percentage of the total LOC above 10%.

It is Wyle opinion as a VSTL that in order to maintain consistency of review across all systems, a minimum value of 5% of total LOC should be manually reviewed. Choosing this minimum value is subjective due to the differences of quality commenting found in various election system software. However, based on the election systems reviewed historically at Wyle VSTL, there has been approximately between 800,000 to 2 million total LOC for a given election system. For a historically typical system, this would amount to a manual review of 40,000 to 100,000 lines of code for the analysis of commenting quality. It is Wyle's opinion that reviewing this amount of code should be sufficient to begin to reveal the pattern of commenting quality, whether excellent or poor or in between; allowing a VSTL to make an engineering decision to stay at 5% for the manual review or increase it for improved evaluation.

The published Java Coding Convention Requirements of Java Code Conventions September 12, 1997 document was used as the Java coding convention standard. The entire source code base was subjected to the automated tool evaluation. In addition, an approximate 10% of the code base submitted was manually evaluated for compliance. The code base evaluated did not fail any of the Java requirements that are written as absolutes during automated evaluation. There was, however, a single instance of a non-enumerated constant discovered; which is a non-absolute Java coding convention. There was also one instance of another non-absolute Java coding convention of indentation styling (found in the manual review process) that was not strictly adhered to. This

7.0 CONCLUSION (CONTINUED)

check is available in Checkstyle, but was apparently disabled. [It should be noted that the indentation examples missed by Checkstyle would not be considered a violation under the EAC 2005 VVSG standards.]

The only Section of the EAC 2005 VVSG that is currently required per RFI 2010-02 is Volume I – Section 5.2.7a. The submitted test source code did not have any section 5.2.7a discrepancies located in the 10% manually reviewed code.

Though not required, the pilot system submitted for testing also had custom Checkstyle configurations that automatically checked for many of the EAC 2005 VVSG standards, as documented in Section 4.0 of this report.

It is Wyle's professional opinion as a VSTL that the code base evaluated in this pilot test does fulfill the intent of the EAC VVSG 2005 – Volume I - 5.2.6a requirement and conforms to RFI 2010-02 requirements thoroughly. The pilot system and code base submitted should be regarded as compliant under the automated tool code review process.

APPENDIX A: JAVA CODING CONVENTION REQUIREMENTS OF JAVA CODE CONVENTIONS SEPTEMBER 12, 1997

Appendix A consists of excerpts with either absolute or near absolute verbiage that Wyle as a lab considered Java Coding Convention "requirements" in the submitted publication (identified in the excerpts below in highlights). These are the requirements used to evaluate the submitted Java source code for compliance to the industry-accepted standard.

6.1 Number Per Line

In absolutely no case should variables and functions be declared on the same line.

Example:

```
long dbaddr, getDbaddr();
```

Do not put different types on the same line.

Example:

```
int foo, fooarray[]; //WRONG!
```

6.2 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyMethod() {

        int int1; // beginning of method block

        if (condition) {
                int int2; // beginning of "if" block
                ...
        }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
    for (int i = 0; i < maxLoops; i++) { ...
```

[NOTE: This one is not absolute due to the "Avoid" word, but will be reviewed as well due to the "do not" wording and the seriousness of the convention.]

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
```

```
...

func() {
        if (condition) {
                int count; // AVOID!
                ...
        }
        ...
}
```

7.1 Simple Statements

Do not use the comma operator to group multiple statements unless it is for an obvious reason.

Example:

```
if (err) {
        Format.print(System.out, "error"), exit(1); //VERY WRONG!
}
```

7.2 Compound Statements

Braces are used around all statements, even singletons, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces

7.8 switch Statements

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

NOTE: the Following couple of cases are not ABSOLUTES due to the "should" word. However, if they are not adhered to, I will mention it in the report.

[NOTE: start "should <absolutes>"]

8.1 Blank Lines

Two blank lines should always be used in the following circumstances:
• Between sections of a source file
• Between class and interface definitions
One blank line should always be used in the following circumstances:
• Between methods
• Between the local variables in a method and its first statement
• Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment
• Between logical sections inside a method to improve readability

8.2 Blank Spaces

All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
        n++;
}
prints("size is " + foo + "\n");
```

[NOTE: end "should <absolutes>"]

9 - Naming Conventions

Variables Except for variables, all instance, class, and
class constants are in mixed case with a lowercase
first letter. Internal words start with capital
letters.

10.4 Variable Assignments

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

Example:

```
if (c++ = d++) {
        ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
        ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps.

Example:

```
d = (a = b + c) + r;
```

should be written as

```
a = b + c;
d = a + r;
```

10.5.4 Special Comments

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.