The Maldives National Journal of Research
Vol. 1, No. 1, June 2013, pp. 7-32

Automatic assessment of Java code

ADAM KHALID, *The Maldives National University*

ABSTRACT *In this study a tool was developed which marks automatically computer programs written by novices in Java. Existing open source static analysers were evaluated in the development. Metrics and marking schemes were developed to mark and grade the programming assignments automatically. A sample of 59 first-year programming students' projects was selected for marking. The projects were marked manually and automatically. The results showed that there is a high correlation between manual and automatic marking for all criteria. The mean Kendall's tau is over 0.75 indicating a very high level of correlation between manual and automatic marking. However, depending on the marking criteria, non-systemic variations were found.*

Introduction

In academic institutions, a mandatory first year unit for computer science majors is a unit entitled 'Introduction to Programming' or similar. In order to learn programming, practice is essential. Practice is provided in timetabled laboratory sessions and in marked assignments. Feedback on both of these is essential. From a student's perspective feedback is not easy to get. A constant complaint from the students is that the laboratory sessions are not long enough, and that the students have to wait for a long time to get the instructor's attention (Tremblay & Labonte, 2003).

For instructors, marking a computer program is time consuming. Academics find marking boring and it is the least liked task for them (Tremblay & Labonte, 2003). In order to mark a program, the instructor has to check for functional correctness as well as the design. To check for functional correctness, the instructor will run the students code against the instructors test cases. To check for design, the instructor has to read the whole code written by the student. Reading and commenting take a lot of time. When the classes are large the problem becomes worse. One way around this problem is to hire more staff but with today's funding this is not a feasible option. If there are many markers there will be a consistency problem and some students will be disadvantaged. If the assignments can be marked automatically or even semi-automatically, the marking load for the instructors will be eased. Automatic marking can be more consistent as well.

In this research the possibilities of automated marking will be analysed and a methodology by which automatic marking can be done will be studied.

ISSN 2308-5959/21030101

Review of Literature

A significant amount of work has been done in the areas of teaching and learning programming, grading programming assignments, automated assessments and analysis of the results. This section explores some of the work that has been done in the areas of teaching and learning, automated assessment, grading programming assignments and analysis of student code.

Programming courses are regarded by many students as extremely hard and the dropout rates are very high (Robins, Rountree & Rountree, 2003). It is believed that it takes ten years for a novice to become an expert programmer (Winslow, 1996). Dreyfus, Dreyfus and Athansiou suggest that in the process of novice becoming an expert the novice has to go through five stages, namely: novice, beginner, competent, professional and expert (Dreyfus et al., 1986). To go from one stage to the other, requires time and experience. The approach made by the students in learning also affects the way they learn. Some students take the programming unit as similar to any other study unit like mathematics or physics. They believe that as long as they attend lectures and read the text book they will be fine. By the time they realize what they are doing is not enough and start programming it is too late. The development of programming skills needs time and practice. If the student starts late it will be difficult for the student to catch up with the programming unit.

In order to learn programming, the student has to learn the syntax of the programming language as well as how to apply the language. Text books mainly talk about the features of a particular language. For students, the main difficulty lies not in the understanding of the syntax of a programming language but in applying (Rist, 1996). For example students can explain what a pointer is, but they find it hard to apply pointers in a program (Lister & Leaney, 2003). Even reading a program and understanding are hard for students (Du Boulay, 1986).

Universities nowadays introduce programming using an object oriented approach. Novices find an object oriented programming language much harder to learn than a procedural language (Lahtinen, Ala-Mutka & Jarvinen, 2005). Wiedenbeck, Ramalingam, Sarasamma and Corritore studied students' comprehension of procedural vs object oriented programs. They asked students to study two similar programs written in Pascal and C++ and were asked questions on programming ease (1999). The results indicate that students found features in C++ were much harder to learn than features in Pascal (Burkhardt, Détienne & Wiedenbeck, 2002).

Student Program Evaluation and Management Systems

A wide variety of software tools exists for managing and assessing student programs. These systems and marking tools can be classified into three major categories based on the aspects of the marking process they support.

(1) Tools on management of distribution and collection of assignments: These tools handle the administrative tasks of the marking process. It includes submission and distribution of the assignments.
(2) Tools on evaluation of correctness: These tools check whether the program compiles, and produces the correct output.
(3) Tools on evaluation of quality: In order to improve the students programming ability it is important to provide feedback on the quality of their programs.

> Quality of a program can be evaluated by examining the program structure such as programming style, complexity measures including coupling and cohesion, the use of constants, choice of identifiers, and comments (Tremblay, Guerin, Pons, & Salah, 2008).

These categories are not mutually exclusive. A given tool can incorporate many of these categories. In the next section some of the existing tools used in marking are discussed.

Marking Programming Assignments

When marking a program the instructor looks for a "good" program defined in terms of several attributes (Moha, Gueheneuc, Duchien, & Meur, 2010). The criteria to measure a "good" program depend on the instructor (Joy, Griffiths, & Boyatt, 2005). Academics and software professionals stress different aspects for program quality. The most common aspects of quality attributes markers look for are listed below (Joy et al., 2005).

(1) Comments in code: Best practice in programming indicates that the code should be well commented so that a third party or even the programmer should be able read and follow the code.
(2) Code style: Programs should have a clear layout, meaningful identifiers and method names; the code should be easy to read.
(3) Correctness of code: The program should compile and behave according to the specifications provided.
(4) Structure: The program should be structured well. The program should have well defined modules and procedures.
(5) Efficiency: The program should be efficient and appropriate language constructs should be used.

Overview of Existing Marking and Managing Systems

In this section some of the existing software developed for marking and managing programming assignments will be discussed.

TRY System. TRY is a software developed at the Rochester Institute of Technology in 1989 (Reek, 1989). Its main aim is to assist instructors in the marking of an introductory graduate course in programming. This software allows students to run the instructors test data. This is done by using a utility in Linux —"setuid". The software also keeps a log of the number of successful tests and failures. The software is designed to mark Pascal programs. TRY does not consider style or design issues.

Game. "Game" is an application developed at Griffith University in 2004. The aim was to mark programs written in Java, C and C++ (Blumenstein, Green, Nguyen, & Muthukkumarasamy, 2004). The software can check commenting, indentation errors and magic numbers. In marking comments it can only count the comments at the top of functions. It cannot check comments for ambiguity in commenting. It marks indentation by setting key points such as beginning of

functions, control statements in a program, and look at the indentation at those key points. In marking the functionality, Game does a text based comparison of the instructors output with that of the students. This is done by using a feature in Linux called "diff." If the student has done the assignment using a different operating system Game will not be able to mark it.

BOSS. BOSS is software developed at the University of Warwick (Joy, Griffiths, & Boyatt, 2005). It can mark programs based on correctness, style and authenticity. BOSS can manage the submission of the assignments.

Functionality of the program is checked by doing a text based comparison of the student output with that of the instructor's test cases. There are drawbacks of this approach. One such drawback is, if the student's output is done on a different system then several non-printing characters will be introduced into the student's output. This means it will not match with the instructor's output.

In marking programming style it can check for the presence of comments, choice of identifiers, layout and efficiency of code (Joy et al., 2005). The program efficiency can be measured by calculating the running time of the program.

OCETJ. OCETJ was developed at the UQAM Canada. The main aim of developing this tool is to provide early feedback to students (Tremblay & Labonte, 2003). In this tool the instructor creates two sets of test cases; one is called the public test suite and the other is called the private test suite. The students, after completing the assignment, will submit a preliminary version of the assignment. The system tests the student's submission on the public test suite specified by the instructor. Feedback on the number of test cases passed and the number of test cases failed are sent back to the student. Once the student is satisfied with the solution, the student submits the final assignment.

This system helps students to produce an assignment which works minimally (Tremblay & Labonte, 2003). When the deadline is reached the final submission will be marked against the private test suite set by the instructor. The private test suite is more complete than the public test suite. The former is hidden from the student. The advantage with this system is that students can provide a solution that is minimally correct. The main drawback of this system is that the student relies too much on the public test suite.

Later, the same University developed another tool called OTO with similar functionality except that it can mark programs written in any language (Tremblay & Labonte, 2003). The developers made it extensible so that new modules can be added to it.

CourseMarker. CourseMarker is another tool developed to mark programming assignments. It is designed as an improvement for Ceilidh developed in 1987 (Higgins, Gray, Symeonidis, & Tsintsifas, 2005). This tool can also handle the submission and marking of programming assignments. This tool, in addition to checking functionality, can check for indentation, length of identifiers to see whether short variable names have been used, and the use of symbolic constants. The tool will allow the instructor to give a grade to an assignment based on the weights the instructor puts for different components of the marking. When marking is complete feedback is provided to the student.

Evaluating Program Functionality and Program Quality

In this research, two aspects of computer programs were evaluated manually and automatically. The two aspects are program functionality (or program validity) and program quality.

Program Functionality or Validity

Program validation is the process of checking whether the program conforms to the requirements. In general validation involves testing. Testing guarantees that the program satisfies its specification, but as noted by Edsger Dijkstra, testing can prove the presence of errors but not their absence (Olan, 2003).

Two approaches are followed in testing. The black-box test approach and white-box test approach. In the black-box test approach the test data is selected solely based on the program requirements for inputs and outputs. In the black box test approach the internal working of the program is not considered. In the white box approach specific knowledge of the programs control flow is considered. In marking a computer program a white box test approach is ideal as this will enable the marker to examine all paths through the student code.

The growth of object oriented programming languages has changed the way programmers test their code. Object oriented programmers favour testing to be based on classes (Olan, 2003). Unit testing is normally followed. Unit testing is based on taking a unit of code and then comparing it with the expected results. In Java a unit is usually a class. Unit tests involve taking one or more methods from a class and these methods are verified automatically (Olan, 2003). When marking a programming assignment unit testing is an ideal thing to do as it will test every method to its expected output. In the following section, program functionality testing is described using JUnit.

Marking by using JUnit. JUnit is a framework for unit testing in Java (Junit, 2010). JUnit is initially developed by XP's (Extreme Programming) proponents (Tremblay et al., 2008). It uses a hierarchical approach to the design and coding of test cases. In JUnit a test class consists of a collection of test suites where each suite consists of a collection of test methods (Tremblay et al., 2008). The main class in JUnit is the Test case. Within this class the programmer defines new methods. These methods contain calls to the application's methods and Assert statements on the results check when the test is run. All the results of the assert statements that are true and that fails will be displayed. The most commonly used Assert statements are listed below.

(1) *AssertEquals:* This method checks whether the two values are equal. If they are not equal the method will raise an error assertFailure message. Example of its usage is shown below.

```
Program 1 AssertEquals an example
Public void testAddTest() {
Int answer =4;
assertEquals((2+2,answer));
}
```

(2) *AssertTrue:* This method will check whether a condition is true. It will throw an exception if the assert fails.

(3) *AssertFalse:* This is the opposite of assertTrue and checks whether a condition is false.

(4) *AssertNotSame:* This method will check whether the two objects do not refer to the same object using the == operator. If they do, it will throw an assertFailed error. Example of its usage is shown below.

```
Program 2 Assert Not Same example
@test
public void testStudentConstructor(){
Student std1 = new Student("Peter",23);
student std2 = new student("Henry",25);
assertNotSame("not the same name",std1.name,std2.name)
}
```

A JUnit test case applied to a simple Java example is shown in the Program Listing 3 and its test case is shown in Program Listing 4. The assert statement given above will return true if the numbers add up correctly and will give false otherwise.

To use JUnit in marking, a JUnit test class can be created. This test class can be used to mark the projects of all the students thus giving consistency in marking. JUnit can be run from the command line. This makes it even easier to do the marking. Scripts can be created so that the same JUnit file is run for all the projects and the results stored in a csv file.

```
Program 3 example Java code for JUnit
public class Calculate {
static public int add(int a, int b){
return a+b;
}
}
```

The JUnit test file for Program Listing 3 is shown in Program Listing 4.

```
Program 4 Junit Test case for the java code
import junit.framework.*;
public class TestCalculate() extends testCase {
public class TestCalculate() {
int num1 = 4;
int num2 = 5;
int total = 9;
sum = Calculate.Add(num1,num2);
assertEquals(num1,num2);
}
}
```

Program functionality or validation is the most important aspect of program evaluation. The second aspect is program quality. This latter aspect is evaluated using a set of measures called style metrics.

Program Quality Measures and Style Metrics

The following are the attributes used in ISO 9126– a widely accepted international standard for software quality: (1) functionality, (2) reliability, (3) usability, (4) efficiency, (5) maintainability, and (6) portability. However, Berry and Meekings (1985) suggested other metrics of style to evaluate program quality.

Berry and Meekings (1985) proposed a style metric, which is based on clarity and understandability of programs. The authors suggest that the program features (namely, module length, identifier length, comments, indentation, blank line length, embedded space, constant definitions, reserved words, included files and GOTO's are indicators of program quality (Mengel, & Yerramilli, 1999). All the features are mapped onto a scale between 0 and100, where 0 is the lowest and 100 is the highest. In analysing the students code metrics were developed based on the following attributes.

(1) *Correctness:* correctness is marked based on the requirements set up by the instructor.

(2) *Style:* module length, identifier length, comments and indentation.

(3) *Efficiency:* CPU time taken by the student program compared to the instructors program

(4) *Complexity:* of the program based on the McCabe's cyclomatic complexity.

Hung, Kwok and Chan evaluated the students' performance based on four software metrics which then later combined to a single one (Hung, Kwok, & Chan, 1993). The four metrics are programming skill, complexity, programming efficiency, and complexity. Hung et al. suggest that the number of lines of code is a good measure for measuring programming skill. McCabe's Cyclomatic complexity metrics are measures for complexity. The program execution time can be used to measure efficiency.

In the style metrics of Berry and Meekings (1985) several factors that are used to assess a computer program were not considered. For example, style is measured based on indentation and length of code. Factors such as variable naming an the use of constants were not considered.

The programs which the students write are short programs and CPU time for these would be negligible and cannot be considered as a measure for efficiency. McCabe's metric is a well-known metric for measuring complexity; however it reports only one number. It does not consider the fact that programs include simple and complex parts.

There are many tasks in the marking for which automation cannot be used as yet. Table 1 shows the tasks in the marking process that can be automated and tasks that can be done manually.

Table 1
Marking Tasks that Can be Automated

No	Task	Fully automated	Semi automated	manual
1	Functional correctness	✔		
2	comments		✔	
3	Variable naming		✔	
4	Indentation and readability		✔	
5	Magic Numbers		✔	

Available tools for code inspection to determine program quality. In this section, some computer-based tools used for analysing and marking student's programs are outlined. Marking is carried out for two purposes: correctness and design. Marking for correctness involves testing the code against the requirements. Marking for design involves checking whether the code is correctly designed and whether the coding conventions set by the programming language have been followed. These two aspects could be determined by code inspection.

Code inspection involves carefully going through the code, design documents and checking for problematic areas. Code inspection is a useful technique to detect potential problems in code. In the industry code inspections has been found to reduce the development cost and increase the software quality (Fagan, 1999). It is estimated that inspections can detect 57% of the defects in code and in design documents (Nagappan, Williams, Hudepohl, Snipes, & Vouk, 2004). It is generally accepted that the cost of repairing a defect is much lower if the defect is fixed early in the development stage than fixed later (Nagappan et al., 2004). Another advantage of code inspection is that software can be analysed before it is tested, potential problems identified, and fixed early, when it is still cheap to fix the problem.

In marking, the marker has to do a code inspection of the entire student's code. Table 2 lists the areas the marker looks for when an inspection is done. Inspection is a long and a tedious process. The marker has to do this for all the projects. If the task is semi-automated the inspection process will become much faster and consistency will be maintained in marking.

Table 2
Java Inspection Check List

Item	Check item
Java doc comments	Java doc comments on all the methods used
	All the return statements explained in comments
	Comments indicating the purpose of all the variables
Variable naming	Class names nouns

	Internal words capitalized
Interface names	Method names verbs, and internal letter capitalized
	First letter of all the variables in lower case and the first letter of all the internal words capitalized
	Meaningful variable names
	All the constants capitalized and words separated by underscores
Java bracketing	Open braces at the end of each declaration statement
	Closing brackets on a new line and indented to match the opening statement
	Loops have braces
Class structure	Classes well structured
	Method lengths not too long
Private and public variables	Code has proper encapsulation and information hiding
Symbolic constants	Code has symbolic constants used rather than magic numbers
Interface and method variables	Correct use of instance and method variables

Static analysis is the checking of the code without actually executing it. There are several open source tools available that can do static analysis. To carry out automatic marking for static analysis, some of these tools can be used. There are two types of static analysis tools: style checkers and bug checkers (Hovemeyer & Pugh, 2004). Bug checkers try to find sections of code that violate the correctness of programs whereas style checkers try to find code that violates the coding standard guidelines. Tools used for this research are discussed below.

PMD. PMD is a BSD licensed (open source) tool developed by Tom Clepland (Seo, Kim, Kim, & Lee, 2009). PMD analyses the Java source code and looks for non-functional and code quality errors. Initially developed as a plug in for Eclipse; a Java programming environment but can be run with any Java programming environment. It can be easily run from the command line as well. PMD has 22 rules. Out of the 22 rules, the rules used for this research are shown on the table in Table 3.

PMD looks for a long list of bad programming practices. Most of the errors PMD picks up are stylistic errors. These stylistic errors are due to bad programming practices and may lead to errors in the future (Christopher, 2006).

Java source code is a text file. The text file, when structured in a certain way, becomes valid Java code. This structure is expressed in a meta-language called EBNF (Extended Backus-Naur Form). This EBNF is referred to as a grammar.

JavaCC (Java Compiler Compiler) reads the grammar and generates a parser that can be used to parse programs written in a programming language (Seo et al., 2009). Another layer JJTree which is an add-on to JavaCC, parses the Java source code to an abstract syntax tree (AST). This tree can be traversed using a visitor pattern (Seo et al., 2009). An example of an abstract syntax tree created for a simple "Hello World" Java code (Program 5) is shown in the Figure 1.

Table 3
PMD Rules

Rule	Description
Basic naming	Basic rules that need to be followed
	Checks for standard Java naming conventions
Braces	Correct use of braces
Code size	Checks for long methods, methods with too many parameters, cyclomatic complexity and n-path complexity

```
Program 5 example pmd
public class HelloWorld {
    public static void main(String[] args){
         System.out.println("Hello World");
}
}
```

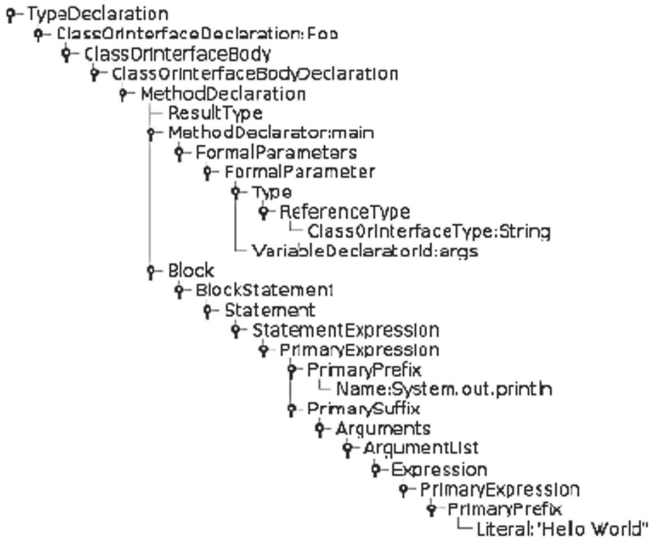


Figure 1. Abstract syntax tree for PMD.

The AST can be represented in XML form. PMD uses XPath styled searches in the AST. Many of the default rules that PMD uses are instances of XPath Rule instantiated with an Xpath that represents a bad structure in the AST (Christopher, 2006).Instantiating an XPath does not require any code; the developer needs to provide the XML file that defines the new rule.

PMD can be used from the command line or a plug-in for Eclipse, or as an Ant element. For this project PMD is run from the command line. When the command is typed with the source file, the following sequence of events follows (Hsu, Jagannathan, Mustehsan, Mwmufiya, & Novakouski, 2007).

(1) In the command the user writes, the location of the Java source file along with the rule or rules that the user wants to execute is/are identified.
(2) PMD reads the Java source file and supplies it to a parser which creates an abstract syntax tree (AST).
(3) The AST is returned to PMD which gives it to the symbol table layer. This layer identifies the scope, the declarations and various usages.
(4) If a particular rule involves a data flow analysis then the AST is given to the DFA (deterministic finite automation) layer which creates control flow graphs and data flow nodes.
(5) With all the data obtained, each rule traverses the AST and detects issues based on the traversal.
(6) All the issues identified are printed to the console or to the type of file specified by the user.

Checkstyle. Checkstyle is another free and open source development tool that helps to ensure that the Java code conforms to the coding conventions established. Checkstyle is commonly used as a plug-in for Eclipse but can be used with any Java programming environment. Checkstyle can be run from the command line. Checkstyle comes with many ready-made coding rules, and allows the user to create his or her own rules as well.

Checkstyle uses an ANTLR parser. The tree corresponds to XML tags. The text column corresponds to the value of the tag, the line and column correspond to the tag attributes (Vashishtha, & Gupta, 2008).

Checkstyle checks are implemented in terms of modules. Modules contain other modules and hence can form a tree structure. An example is shown in Program Listing 6.

Program 6 Checkstyle original example

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
   "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
   "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
     <module name="TreeWalker">
         <module name="ConstantName"/>
         <module name="LocalFinalVariableName"/>
         <module name="LocalVariableName"/>
    </module>
</module>
```

Checkstyle checks are based on a configuration file which is in XML format. The components of Checkstyle parse the instructions in the configuration file and check the input source file against the configuration file. The results of the source code checking will be output in the specified format. The output can be "written" to the console or as an XML file.

Checkstyle can be tailored to find the errors that the user is looking for. For example, if the user wants to list only the magic number errors in the Java source code, the XML file provided by Checkstyle can be modified to detect only the magic number (symbolic constants) errors in code.

Grading of programs based on evaluation of functionality and quality

A grade given to a programming assignment or project reflects how well the student has met the objectives of the assignment. The purpose of grading is to discriminate the functional correctness and quality of the programs written by different levels of students. In this research four levels were chosen as greater discrimination of different attributes would make the research more cumbersome. The four levels chosen correspond to "Excellent," "Good," "Satisfactory" and "Poor." The grades and the achievement levels for the grades are listed below.

(1) *Excellent:* The student has fulfilled all the objectives of the assignment and has gone beyond expectation.
(2) *Good:* This grade will be given to students whose programs fulfil the requirements and have not gone beyond expectation.
(3) *Satisfactory:* This grade is given to students who are close to meeting the expectations of the program.
(4) *Poor:* This grade is given to students who are below the expectations for the assignment.

Grading Functional Correctness. Functional correctness refers to how well the requirements are met in the program. The grade given to a program for functional correctness reflects the number of defects in the program. One common metric to measure defects is in Equation 1 where NCLOC is the non-commented lines of code. This equation measures the number of defects per line of code.

$$PercentDefectsPerLOC=(\sum(defects))/NCLOC\times 100 \quad (1)$$

In marking functional correctness two groups of test data are selected. The easy test cases are simple test cases that should be achieved based on the objectives of the assignment and hard test cases are harder parts of the code to program.

Equation 2 is used to get the percentage tests passed for each of the two groups. The results obtained from the Equation 2 should be mapped on to a grade. Table 4 maps the results from the Equation 2 on to a grade.

$$PercentTestsPassed=(\sum(testsPassed))/(\sum(testcases))\times 100 \quad (2)$$

Table 4
Functional Correctness Marking Scheme

Grade	Criteria
Excellent	100% of the easy and hard test cases passed for all the three programs.
Good	100% of the easy tests and for hard tests, the pass rate is between 75% and 99% at least 75% of the hard test cases passed.
Satisfactory	100% of the easy test cases and at least 50% of the hard tests passed
Poor	If any of the above criteria is not met.

As Table 4 indicates, to obtain an "Excellent" grade, the student has to demonstrate that he has achieved more than the expectation. This represents a 100% pass rate for all the test cases chosen. In designing the marking schemes the amount of time available to do the assignment is also considered.

Grading for Program Quality. From the literature reviewed previously, it is noted that the following attributes are used to measure software quality: (1) commenting, (b) variable naming, (c) lines of code, (d) indentation and bracketing, (e) use of symbolic constants, (f) cyclomatic complexity, and (g) duplicate code. For evaluating student programs in this research, cyclomatic complexity and duplicate code analyses were not carried as they would not be meaningful for the simple projects used in the sample.

Commenting. Commenting is a discipline that students need to master when they learn programming. Comments help the reader in understanding the code. Java has got conventions for commenting code. Students are expected to follow the Sun Java coding conventions. The marking scheme developed to grade comments is shown in Table 5.

Table 5
Marking Guide for Comments

Grade	criteria
Excellent	The program has got comments for all the public classes and methods and has got appropriate @ statements for returns, parameters and exceptions. A comment present for all the instance variables explaining their purpose. All the comments follow the sun-java coding conventions. All the comments are meaningful
Good	The program has got comments missing for less than 15%of the methods.

	Less than 25% of the instance variables not commented.
	Comments follow Java commenting rules.
	Comments are meaningful.
Satisfactory	The program has got comments missing for about 25% ofthe methods
	Comments written are meaningful.
	Follows Java commenting conventions.
Poor	The above conditions are not met

For evaluating commenting, counting the number of comments alone will not be a good measure. Factors like the length of the code or the number of methods and variables have to be considered. Two metrics can be used to measure comments. They are shown in Equations 3 and 4 where NM is the number of methods and NVar is the number of variables and NF is the number of fields and CLOC is the number of commented lines of code. In this research, Equation 3 is used.

$$commentDensityPerMethod=CLOC/((NM+NVar+NF))\times 100 \quad (3)$$

$$commentDensityPerLOC=(\sum CLOC)/LOC\times 100 \quad (4)$$

Automatic marking of comments. In automatic marking the marking tool counts the number of violations in commenting. These numbers needs to be mapped on to a grade. Cut-off points need to be determined. The cut-off points should determine the number of violations acceptable for an "Excellent" grade and number of violations acceptable for a "Good" and so on. Table 6 gives the cut-off points used to mark comments used in this research for grading commenting.

The values are obtained using a tool: JavaNCSS. JavaNCSS gives the number of comments per method. This is what is used in this study. In manual marking the statistic is obtained from the Equation 4. The cut-off points for the grades can be obtained from Table 6.

Table 6
Commenting cut-off points

Grade	criteria
Excellent	CP =100
Good	85 < =CP <100
Satisfactory	75 <= CP < 85
Poor	CP < 75

Variable Naming. Variable naming is another important aspect of program quality. Using appropriate variable names will help the programmer and others who read the code to understand the code better. Sun Java has several variable

naming conventions which the programmers have to follow. Students are expected to write code with variable names that are in line with Sun Java conventions. The marking scheme adopted to grade naming errors is given in the Table 7.

Table 7
Variable Naming Cut-off Points

Grade	criteria
Excellent	Meaningful and appropriate names for over 95% of the code
Good	Over 85% of the code has used appropriate variable names and are in line with Sun Java coding conventions.
Satisfactory	Over 75% of the code has used appropriate variable names and are in line with Sun Java coding conventions.
Poor	If any of the above criteria is not met.

In order to automatically mark and grade variable naming a marking tool, called PMD, is used. It counts the number of variable naming errors. It can detect variables that are too short and variables that are not in line with Java variable naming conventions. In measuring violations in variable naming, the length of the program and the number of variables have to be considered.

The following two equations can be used to measure the percentage of naming violations. Equation 5 and Equation 6 give the percentage violations. In the equations, NCLOC is the number of non-commented lines of code. In this research, Equation 6 is used. For automatic marking Equation 6 is mapped on the marking scheme in Table 6.

$$namingErrorsPerLoc = numberOfNamingViolations / NCLOC \times 100 \quad (5)$$

$$namingErrorsPerVariable = Violations / (\sum(Variables)) \times 100 \quad (6)$$

Indentation and Code Readability. Code indentation is another area where students need to master when they learn programming. Code indentation and bracketing make the code more readable and hence easier to maintain and redevelop. Java has conventions for indenting, which students are expected to follow. In calculating indenting violations the number of methods is taken in to account. As students are expected to indent every method, Equation 7 is used to measure indent violations.

$$IVPerMethod = (\sum indentErrors) / NumberOfMethods \times 100 \quad (7)$$

IVPerMethod is the indent violations per method. The marking scheme shown in the Table 8 is used to mark indentation and bracketing when manual marking is used. IVPerMethod is the indent violations per method.

Table 8
Indentation Violations

Grade	Criteria
Excellent	Main sections of the program are easy to follow. The code is indented properly and has used correct usage of braces followed
Good	85% < IVPerMethod ≤ 100%.
Satisfactory	75% < IVPerMethod ≤ 85%
Poor	Above criteria are not met.

Magic Numbers. Magic Numbers are numbers that are hard-coded in the program. If numbers are hard-coded in the program then if a change is required the programmer has to find all the hard-coded numbers and make the required changes. The use of magic numbers is considered bad programming style and is discouraged in programming. Students will be marked down if magic numbers are used instead of symbolic constants. The marking rubric used to assess magic numbers is given in Table 9.

Table 9
Magic Numbers Marking Criteria

Grade	Criteria
Excellent	No Magic number used at all
Good	There are magic numbers used but is insignificant
Satisfactory	Magic numbers used but has used symbolic constants as well.
Poor	None of the above criteria is not met.

In counting magic number violations Checkstyle is used. Checkstyle counts the number of violations. In counting magic number the code length is considered. In deciding on an equation to measure magic number usage, a few trial programs were run. It is discovered that the magic number usage in student programs is high when the programs had displays. In the equation a correction factor, CF, is used to account for displays. CF depends on the display the program is using. In manual marking the instructor can decide on CF. Equation 8 gives the percentage of magic number violations per Non-Commented Line of code (NCLOC).

$$magicNumbersPerNCLO=(\sum magicNumberViolations)/NCLOC\times CF\times 100 \quad (8)$$

Lines of Code. The number of lines of code is the most common measure to estimate effort in software (Fenton, & Pfleeger, 1997). The metric is defined in many ways. Sometimes, programmers use blank lines to make the program more readable. If blank lines are counted when measuring lines of code then the metric will not be a good estimate of programming effort. This is true for comment lines as well.

In counting lines of code, the Hewlett-Packard's definition for lines of code is widely used (Fenton, & Pfleeger, 1997). In this definition, comments are not

considered as lines of code. For non-commented lines of code the abbreviation NCLOC issued. This is called EFLOC (Effective lines of code). The abbreviation used to measure the number of commented lines of code is CLOC. With these abbreviations and Hewlett-Packard's definition for lines of code or the total length of a program is:

$$LOC=NCLOC+CLOC \tag{9}$$

Lines of code (LOC) can be used to measure how well the student has written the program. If the student's program is too short, this is an indication that some functionality is not implemented. On the other hand, if the code is too long, then it is an indication that the student had not implemented some feature, such as loops, properly.

To create a marking scheme, that is, to mark and give a grade, the estimated correct size of the program has to be established. One way to get the correct size of the program is for the instructor to write a model solution. Assuming the instructor's solution is the correct solution, it may be used as a benchmark measure. The other alternative would be to go through the student programs or projects and pick a project which is well done and fulfils all the requirements. In this research, the latter method is used. The equation used to measure the percentage lines of code based on the model solution is shown as Equation 10. Figure 2 shows the ranges and cut-off points for grading lines of code.

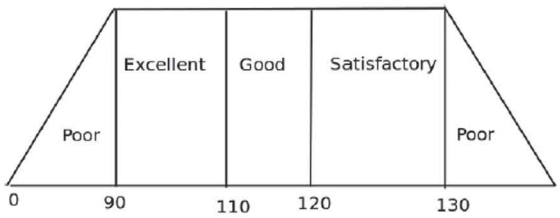


Figure 2. Lines of code cut-off points.

$$LOCRatio=studentsNCLOC/solutionSampleNCLOC\times 100 \tag{10}$$

Based on Equation 10 a marking scheme was developed, which is shown in Table 10.

Table 10
LOC Marking Criteria

Grade	Criteria
Excellent	90 <LOCRatio< 110
Good	110 <LOCRatio<120
Satisfactory	120 <LOCRatio<130
Poor	Above criteria not met

Cyclomatic Complexity Cyclomatic complexity refers to the number of independent logical paths in a program. It also establishes the maximum number of different test cases required to ensure that each program statement is executed at least once (Spinellis, 2006). Simple programs with a sequence of statements will have a cyclomatic complexity of 1. If the program has got case labels, the cyclomatic complexity will be increase by 1 for each case. For a program with a flow graph the cyclomatic complexity is calculated as in Equation 11.

$$M=E-N+P \tag{11}$$

Where M is the cyclomatic complexity, E is the number of edges in the graph and N is the number of nodes and P is the number of connected components. The programs used in this research were simple and short programs where cyclomatic complexity will not vary much and hence it is not a metric used in the marking.

Marking and weighing individual components
In this study several features of the program have been measured. To come up with an overall grade for the program, the individual components need to be converted to a total score. For an overall score, weights for each component have to be assigned. Table 11 shows the weights.

Table 11
Weight for Individual Components

No.	Attribute	Weight
1	Comments	4
2	Naming	2
3	Indentation and Bracketing	3
4	Usage of Symbolic Constants	3
5	Method Decomposition	4
6	Code Length	3

To obtain an overall grade for the code, the grades obtained for the individual components need to be combined using some scale factors. To arrive at a final grade Equation 12 is used. The scale factors are shown in Table 12.

$$grade=\sum scalefactors \times Weight \tag{12}$$

Table 12
Scale Factors

Excellent	Good	Satisfactory	Poor
4	3	2	1

Method

The experiment was carried out using the first-year programming projects written by the students of the University of Western Australia in 2007. The students were required to write three Java classes: Weather Station Class, Weather Display Class and Weather Centre Class.

The Weather Station Class. This program requests items of information from the user. The Weather Station stores maximum, minimum temperatures and rainfall for one whole year. The constructor of the Weather Station should have signatures for name, maximum, minimum, and rainfall, where name is a string and maximum, minimum and rainfall are arrays of type double. The students are required to do error handling such that the array sizes should be exactly equal to twelve.

The Weather Display Class. The Weather Display class should plot graphs with labelled axes for the maximum and the minimum temperatures, and rainfall for the Weather Station. The display should be a line graph for the maximum and minimum temperatures, and a bar chart for rainfall. One side of the y-axis should be labelled in degrees Celsius and the other side in degrees Fahrenheit.

The Weather Centre Class. In this program students are required to handle several Weather Stations. When called, the program should display two windows, one window displaying data for the current Weather Station and the other showing a list of ten available Weather Stations. The array should be sorted so that the current weather station should be at the front of the array.

Sample

From the sample of 61 projects available to the researcher, a sample of 59 projects was selected for this experiment. Two were found to have compiling errors and were excluded. The selected 59 projects are the projects that compile, and are from students who had completed the three Java classes required.

Procedure

In this experiment two types of marking were used: human and automatic. Both markings were done by the author. It would have been better if the manual marking was done by another person. However, it was not possible to get the services of another marker due to cost. In order to reduce bias obtained from one type of marking to the other, manual marking was done first. The automated marking is done by writing a program in Python and running the program for all the students' codes. The results obtained from automatic marking and manual marking were analysed.

Manual marking was done by writing test cases and executing each and individual project for the test cases. The numbers of successful and failed tests were recorded. Marking for code design is done by carefully reading through the code and identifying potential problems.

Correctness. Testing for correctness or functionality requires the instructor to write test cases and run it for all the three programs and note the number of test cases passed and failed. In order to mark functionality, two sets of test data were selected; one set of data for parts easy to implement and one for those hard to implement. Parts easy to code were simple functions which most students should

be able to complete without much difficulty. Parts that are difficult to code are harder and many would have found it challenging. Marking and grading were done based on Equation 13 and Equation 14.

$$EasyTests=(\sum EasyTests)/(\sum TotalEasyTests)\times 100 \quad (13)$$

$$HardTests=(\sum HardTests)/(\sum TotalHardTests)\times 100 \quad (14)$$

Based on Equations 13 and Equation 14 a grading rubric was created (Table 13). The values are set by the instructor and can be changed from assignment to assignment.

Table 13
Grading Rubric for Functionality

Grade	Easy	Hard
Excellent	Number of Tests passed = 100	Tests passed = 100
Good	Number of Tests passed = 100	Tests passed > 85
Satisfactory	Number of Tests passed = 100	75 <tests passed< 85
Poor	Above criteria not met	Above criteria not met

Once the test data is fed to the program, marking was done automatically and the defect density per line of code found is shown in the Figure 3.

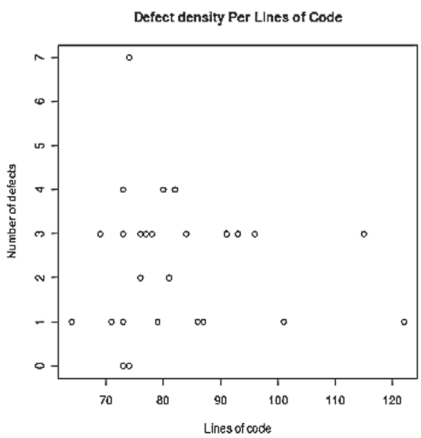


Figure 3. Defect density against lines of code.

Coding Style. In this section the following attributes of code quality is marked and graded.

Comments. Comments are marked using a tool, JavaNCSS. JavaNCSS gives the number of non-commented methods. Markers generally look for the

commenting violations in Java and also the meaningfulness of the comments. The meaningfulness of the comments cannot be identified and this attribute had to be marked and graded manually.

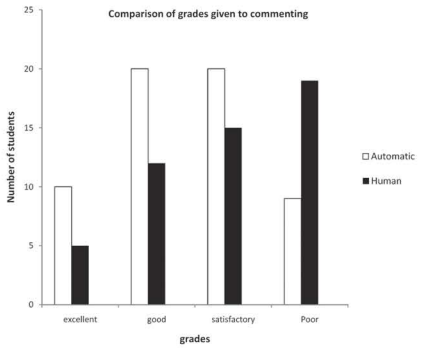
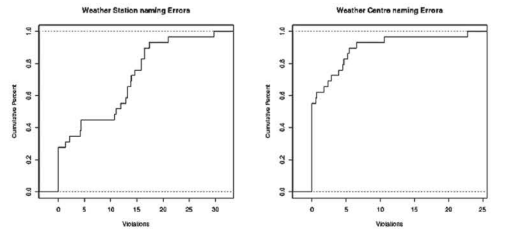


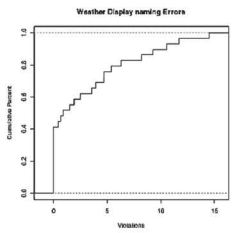
Figure 4. Bar graph comparing the manual and automatic marking for comments.

Variable Naming. Variable naming is marked by using PMD which identifies all the variable naming violations used by Java. However, the meaningfulness of the variables should be marked manually. Figure 5 shows the cumulative distribution of the naming violations.



(a) Weather Station: Naming Cumulative Distribution

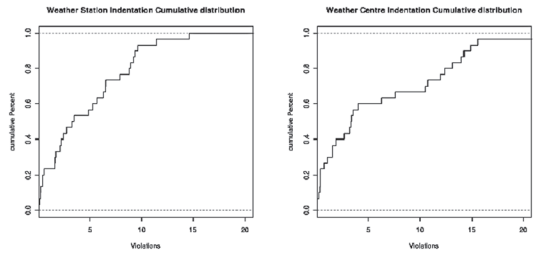
(b) Weather Centre Naming Errors Cumulative Distribution



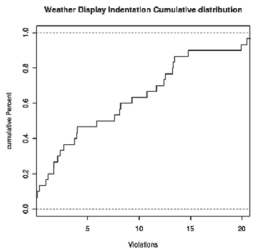
(c) Weather Display Naming Cumulative Distribution

Figure 5. Cumulative distribution of naming violations.

Indentation. Indentation helps the reader to read and follow the code. For this research indentation errors are calculated using JavaNCSS. The following figure (Figure 6) shows the indentation violations made by the students.



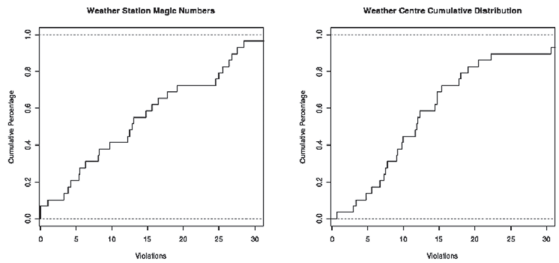
(a) Weather Station: Indentation Cumulative Distribution (b) Weather Centre Cumulative Distribution



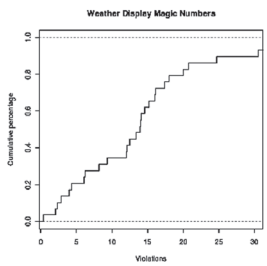
(c) Weather Display Indentation Cumulative Distribution

Figure 6. Indentation violations of student programs.

Magic Numbers. Magic numbers are hard coded numbers in the program. Magic number violations were marked using CheckStyle. Figure 7 shows the magic number violations for the three programs marked.



(a) Weather Station: (b) Weather Centre



(c) Weather Display

Figure 7. Magic number violations for the three programs.

Discussion

In this study we have developed a tool that can marks programs written in Java. We have compared the results of marking using manual methods and automated methods. The results are analysed using statistical methods. Table 14 summarizes the results obtained for both marking.

Table 14
Comparison of Results of Manual and Automatic Marking

Attribute	Weather Station project	Weather Center project	Weather Display project
	Kendall tau	Kendall tau	Kendall tau
Commenting	0.906	0.876	0.675
Variable naming	0.783	0.569	0.796
Lines of code	0.730	0.777	0.729
Magic numbers	0.879	0.950	0.849

To compare the results of the two types of marking, Kendall's coefficient tau (τ) is used. Kendall Coefficient tau indicates the correspondence between two values. The result of computing this statistic will give a value between -1 and 1. A value of 1 indicates that the values are in complete agreement and a 0 indicates that the values are not correlated at all. The Kendall's tau is similar to the more well-known Pearson's and Spearman's correlation coefficients. The main advantages of using Kendall's tau are that the distribution of Kendall's tau has better statistical properties and that there is a direct interpretation of Kendall's tau in terms of the probabilities of observing the agreeable (concordant) and non- agreeable (discordant) pairs. Further, in most situations, the interpretations of Kendall's tau and Spearman's rank correlation coefficient are very similar and thus invariably lead to the same inferences (Conover, 1980).

The statistical analysis shows that there is a strong between automatic and manual marking. The tau is highest for Weather Station Project, especially for commenting (0.906). It is the least for variable naming in the Weather Center Project (0.569). For commenting, the Weather Station and Weather Centre shows a high correlation whereas the Weather Display shows a relatively low correlation. The Weather Display is a graphic program in which there are a lot of functions and choice of colours. Thus the number of comments required is a lot more. This is probably one reason why the correlation is less. Variable naming, lines of code and magic numbers all show high correlations between both markings.

Conclusion

In this research, a new grading approach was developed to grade programs written in Java. The normal 5 level grading used by most universities, "HD", "DN," etc., is changed to a four level grading system. This is done in order to automate the grading and make the cut-off points between the grades simpler. The grades used in this research were "Excellent", "Good", "satisfactory" and "Poor."

Marking and grading computer programs were done in two parts; namely marks for design and marks for correctness or functionality. Universities put a lot of emphasis on these two parts. Even if a program works correctly but is poorly designed, it is still considered a poor quality program.

Marking for correctness or functionality involves the instructor writing the test cases and running the student's code with the test cases. Functionality was marked by using JUnit. A script was written in Python and all the student projects were executed using the Python script. The output was sent to a .csv file. The equations and rubrics developed in this research were used to map the values onto a grade.

The marking tool developed cannot fully mark the assignments. However a large part of the marking process can be automated. The human marker, after the automatic marking, needs to spot check the assignments.

This research shows that there is a high correlation between manual making and automated marking and hence the marking process can be semi-automated, thus saving a lot of marking time of the instructors.

Limitations and Future Work

Firstly, the marking schemes developed is experimented only with programming projects of one group of students—the 2007 first-year students' projects. It could have been used for the projects of several more years for validation. Secondly, the sample data is limited to 59 students and three programs. A larger sample would improve the reliability of the results. The marking schemes developed were used only to mark a very basic programming assignment. In this research, validating the marking schemes was not done. The validation of the marking schemes and increasing the sample size are suggestions for further research.

Acknowledgements

I would like to thank Rachel Cardell-Oliver and Terry Wooding of the University of Western Australia for supervising this work. I would also like to thank the anonymous referees and Hassan Hameed, who reviewed this paper and gave valuable suggestions.

References

Berry, R. E., & Meekings, B. A. E. (1985). A style analysis of C programs. *Communications of the Association for Computing Machinery*, *28*(1), 80–88.

Blumenstein, M., Green, S., Nguyen, A., & Muthukkumarasamy, V. (2004). An experimental analysis of GAME: A generic automated marking environment. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE '04) (pp. 67–71). UK, Leeds.

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task, and phase. *Empirical Software Engineering Journal*. *7*(2), 115–156.

Christopher, C. N. (2006, May 10). Evaluating static analysis frameworks. Retrieved from http://www.cs.cmu.edu/~aldrich/courses/654/toolschristopher-analysis-frameworks-06. pdf

Conover, W. J. (1980). *Practical Non-Parametric Statistics* (2nd ed.). New York: John Wiley and Sons.

Dreyfus, H. L., Dreyfus, S. E., & Athanasiou, T. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer*. New York: The Free Press.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.

Fagan, M. E. (1999). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, *38*(2–3), 258–287.

Fenton, N.P., & Pfleeger, S. (1997). *Software metrics: A rigorous and practical approach*. (2nd ed.). Boston: PWS Publishing, International Thompson Computer Press.

Higgins, C. A., Gray, G., Symeonidis, P., & Tsintsifas, A. (2005). Automated assessment and experiences of teaching programming. *Journal of Educational Resources in Computing*, *5*(3), 5–25.

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Notices*, *39*(12), 92–106.

Hsu, A., Jagannathan, S., Mustehsan, S, Mwmufiya, S, & Novakouski, M. (April, 24, 2007). *Analysis Tool Evaluation*: *PMD Final Report*. School of Computer Science, Carnegie Mellon University. Retrieved from: http://www.cs.cmu.edu/~aldrich/courses/654/tools/hsu-pmd-07.pdf

Hung, S, L., Kwok, L. F., & Chan, R. (1993). Automatic programming assessment. *Computers and Education*, *20*(2), 183–190.

Joy, M., Griffiths, N., & Boyatt, R. (2005).The BOSS online submission and assessment system. *Journal on Educational Resources in Computing*, *5*(3), 1–28.

JUNIT(2010). Retrieved from http://junit.sourceforge.net/

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. (2005). A study of the difficulties of novice programmers. In ITiCSE '05: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Education* (pp.14–18), New York: ACM Press.

Lister, R., & Leaney, J. (2003). First year programming: Let all the flowers bloom. In Greening, T. and Lister, R., (Eds.) *Proceedings of the Fifth Australasian Computing Education Conference* (ACE 2003) (pp. 221–230). Adelaide: ACSm .

Mengel, S., & Yerramilli, V. (1999). A case study of the static analysis of the quality of novice student programs. Paper presented at Thirtieth SIGCSE Technical Symposium on Computer Science Education, New York.

Moha, N., Gueheneuc, Y. G., Duchien, L., & Meur, A. F. L. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*, 20–36.

Nagappan, N., Williams, L., Hudepohl, J., Snipes, W., & Vouk, M. (2004, November). Preliminary results on using static analysis tools for software inspection. Paper presented at Fifteenth IEEE International Symposium on Software Reliability Engineering (ISSRE 2004), (pp. 429–439), St. Malo, France.

Olan, M. (2003). Unit testing: Test early, test often. *Journal of Computing Sciences in Colleges*, *19*(2), 319–328.

Reek, K. A. (1989). The TRY system or how to avoid testing student programs. Paper presented at the Twentieth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '89) (112-116).

Rist, R. (1996). Teaching Eiffel as a first language. *Journal of Object-Oriented Programming*, *9*, 30–41.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2).137–172.

Seo, B., Kim, D., Kim, Y. G., & Lee, S. (2009). Mini-Project: Tool or analysis practicum. Retrieved from http://www.cs.cmu.edu/~aldrich/courses/654/tools/theCruX-PMD-2009.pdf

Spinellis, D. (2006). *Code quality: The open source perspective*. Boston, MA: Addison-Wesley.

Tremblay, G. and Labonte, E. (2003, July). Semi-automatic marking of Java programs using JUunit. In the *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications* (EISTA'03) (pp. 42–47). Orlando, FL: International Institute of Informatics and Systemics.

Tremblay, G., Guerin, F., Pons, A., & Salah, A. (2008). Oto, a generic and extensible tool for marking programming assignments. *Software: Practice and Experience*, *38*(3), 307-33.

Vashishtha, S., & Gupta, A. (November 25, 2008). Automated code reviews with Checkstyle Part 1. Retrieved from: http:// www.javaworld.com/javaworld/jw-11-2008/jw-11-checkstyle.html.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S. & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*, 255–282.

Winslow, L. E. (1996). Programming pedagogy - A psychological overview. *SIGCSE Bulletin*, *28*(3), 17–22.