ISSN: 2167-1435
v30 n2537

Teaching Programming Style with Ugly Code

Kirby McMaster
kmcmaster@weber.edu
Computer Science, Lake Forest College
Lake Forest, IL 60045, USA

Samuel Sambasivam
ssambasivam@apu.edu
Computer Science, Azusa Pacific University
Azusa, CA 91702, USA

Stuart Wolthuis
stuart.wolthuis@byuh.edu
Computer & Information Sciences, BYU-Hawaii
Laie, HI 96762, USA

Abstract

In this paper, we describe how good programming style can contribute to better software. Good style makes source code easier to read and understand, which can reduce errors and simplify maintenance. We discuss several popular style practices. We then introduce a software product we have written called UglyCode, which allows instructors to demonstrate various programming style options. Specific examples that illustrate the use of UglyCode follow. With UglyCode, programming style can be visualized interactively, showing the immediate effect of style choices on the readability of code.

Keywords: programming style, layout, ugly code, algorithm, Java.

1. INTRODUCTION

Teaching computing students how to become professional programmers involves substantially more than helping them learn the syntax of a programming language. In a beginning programming class, the focus is on teaching a computer how to solve a problem (Shustek, 2009). This involves a description of a higher-level programming language (e.g. C++, Java, Visual Basic, or Python), and practice organizing the language statements into a working program. At this initial stage, students write code that the computer can understand.

As students gain programming experience, they become more concerned with the design and implementation of algorithms (Dijkstra, 1971). Three desirable characteristics of algorithms receive early emphasis: (1) correctness, (2) performance, and (3) efficiency. We describe these characteristics in the context of the courses that spend substantial class time on them: Data Structures, Algorithms, and Software Engineering. We also discuss the concepts of modularity and maintainability, which have become increasingly important in software development as systems have grown in size and complexity.

ISSN: 2167-1435
v30 n2537

Eventually, to become a professional programmer, a student must be able to develop systems that satisfy additional objectives, such as dependability, reliability, safety, security, usability, and portability.

Correctness

Programmers continually strive to write programs that contain no errors. There are several aspects to program correctness. The aspect that receives the most attention from mathematically-trained computing professionals is *logical* correctness. The primary method used to determine logical correctness is proofs. A software development group in Australia (Klein, 2009; Klein, et al, 2009) recently announced that they have proven their microkernel operating system code to be correct. (It took several years to complete the proof.) Textbooks on algorithms demonstrate proofs for many common algorithms (Cormen, et al, 2009; Sedgwick & Wayne, 2011).

When you prove that an algorithm is correct, the proof does not guarantee that the source code will be without errors. Proofs also do not ensure that the program meets customer requirements, since requirements are often unstated, inaccurate, or changeable.

A proof is based on a mathematical model, which is an abstraction of a real world situation. If the model does not accurately represent the real world, then the proof is irrelevant. In addition, the mathematical model never completely matches the version of the model presented to the computer. For example, the math model may assume that variables are continuous, whereas all numerical values in a computer are discrete.

An alternative approach to verifying program correctness is based on *empirical* results. This approach depends on thorough testing of the software as it executes. A well-designed test plan consists of a broad range of tests, both for individual parts of the system and for the system as a whole (Somerville, 2011). In the Software Engineering course sequence, students should be required to construct test plans for their software development projects before the source code is written.

Performance vs. Efficiency

Programmers are encouraged to write programs that perform well and make efficient use of computer resources, especially the CPU and memory (Silberschatz, et. al., 2012). There is a fundamental trade-off between performance and efficiency. A process can be given dedicated CPU time and "unlimited" memory to improve performance, but at the expense of efficiency.

In a system where multiple programs run concurrently, primary responsibility for managing these tradeoffs is handled by the operating system. CPU scheduling algorithms interleave CPU time with I/O activities. Virtual memory management schemes allow for dynamic reassignment of memory for processes during execution. The goal is to balance *throughput* and *response time* measures for performance.

A programmer can influence performance and efficiency through the choice of *algorithms*. For example, the speed of a sorting algorithm can depend on the type of data being sorted, along with the amount of memory available. Merge sorts are faster when additional memory can be allocated to hold intermediate results (Lafore, 2003).

Often, performance is affected most by a bottleneck in the system. If the slowest part involves disk storage, then memory caching for disk reads can greatly improve performance. Sometimes a small section of code can slow down the system, if performed repeatedly. Rewriting the code in a faster language (e.g. C or assembly language), using multiple threads, or finding a better algorithm can improve performance.

Modularity and Maintainability

Programmers are introduced to modular code in their first programming class (Lewis & Loftus, 2011; Liang, 2012). The modules, in this case, are functions and procedures. Not all of the benefits of modularity are grasped initially. In object-oriented programming, the design and use of classes, objects, and encapsulation becomes a valued way to manage complexity in larger programs.

Maintainability is a less understood characteristic of software. It depends on a variety of methods that make code easier to correct and modify. Most software is not maintained by the original developer. Developers move on, but code will often last for years. Readability is essential for continual maintenance. The Department of Defense estimates that 60-80% of software life cycle costs are for maintenance.

ISSN: 2167-1435
v30 n2537

Modular code is easier to maintain, but other practices can also improve maintenance efforts. Programming courses spend little time directly on maintainability. More detailed presentations appear in Software Engineering books (McConnell, 2004; Somerville, 2011). Topics relevant to maintenance include agile development, configuration management, version control, and refactoring.

What About Programming Style?

Programming style involves ways that a programmer can organize and present code to make it more understandable to other programmers.

"The smaller part of the job of programming is writing a program so that the computer can read it; the larger part is writing it so that other humans can read it." (McConnell, 2004).

This includes a variety of code layout, formatting, and content enhancing techniques, such as the use of white space and variable naming conventions.

By making code more understandable, style improvements contribute to other desirable program features. For example, readable code is more likely to be correct when initially written, and it is easier to modify when changes are required. Programming style can also improve software testing to verify program correctness.

The remainder of this paper covers programming style concepts, our UglyCode software, and style examples in sample code. Section 2 describes style concepts that involve source code layout (e.g. curly braces), along with the insertion of additional content into the code (e.g. meaningful comments). In Section 3, we introduce our UglyCode software environment, which can be used by instructors to show the effect of different style choices on code readability. Section 4 presents several programming style examples that can be demonstrated in class using UglyCode.

2. PROGRAMMING STYLE

The primary purpose of programming style is to make it easier for programmers to understand what the code is doing. According to McConnell (2004), the Fundamental Theorem of Formatting should be: "Good visual layout shows the logical structure of a program."

But which programming style is best? Expert programmers almost always have their own preferred style for writing code. A conversation about which style is best often takes a religious tone. One point of consensus is that "the details of a specific method of structuring a program are much less important than the fact that the program is structured consistently" (McConnell, 2004).

In *The Elements of Programming Style*, Kernighan and Plauger (1978) describe many style choices for programmers. We present a partial list of their style topics. Our discussion of programming style is organized into two groups, layout and content.

Program Layout

Program layout involves techniques to rearrange source code to make it more readable. No content is added to the code, other than changes in spacing. Several layout methods are described in the following paragraphs.

Blank lines: In a written report, blank lines are added between paragraphs and sections to make the report easier to read. Similarly, blank lines can be added to source code between functions and to bind together lines of code that perform some computing activity (e.g. input). The effect for the code is the same as for the report. It makes the code easier to understand. However, too few blank lines, too many blank lines, or blank lines in improper places can disguise the logical structure of a program.

Indenting: Another way to visually present lines of code that "belong together" is to use the same level of indentation for the lines. For example, the code within a loop can be indented a fixed amount. Because source code can have nested blocks of statements, more than one level of indentation can be helpful. Indenting is also used to indicate that a statement wraps over more than one line.

One question that always generates a mixture of responses is "how many spaces to indent?" If tabs are used to indent, then the question is "what tab setting?" Each programmer will have a preferred answer, and many software development environments provide explicit standards.

White space: Blank lines and indenting (with spaces or tabs) use white space. In this

ISSN: 2167-1435
v30 n2537

paragraph we refer to the use of blank characters to provide spacing *within* a statement to improve readability. For example, white spaces can be used to separate the variables in the parameter list of a function. Programmers often exercise unstated habits in their use of white space within statements.

Block layout: A block of code is a sequence of statements having the behavior that either all statements are executed, or none are. In a conditional (e.g. if) statement, the block is executed only when the condition is true. In iterative (e.g. while) statements, the block will be executed repeatedly until the continuation condition becomes false. An important part of block layout is placing marks in the code where each block begins and ends.

Formatting conventions for blocks depend on the programming language. In a language with fully-bracketed syntax (e.g. Visual Basic, with If ... End If), the statements include markers for the start and end of blocks. Recent languages such as C, C++, and Java use curly braces (e.g. "{ ... }") to mark blocks (Kernighan & Ritchie, 1988). There are differences of opinion on how to format curly braces, such as whether or not to put each curly brace on its own line.

Statement length: In assembly language programming, instructions are short, so each instruction easily fits on a single line. Early fixed-format higher level languages such as FORTRAN and COBOL were designed with punched cards in mind (maximum of 80 characters per card). In these languages, a statement will continue across more than one card (line) only if marked in a special way.

Many recent languages are free-format, in that a statement can continue across multiple lines until a termination character occurs (e.g. ";" for Java). The programmer has a choice of how wide to format each line of code, using multiple lines as needed for individual statements. Also, several short statements can be placed on the same line. The selected width may be guided by screen display size and/or printer width. However, very long lines can be as unreadable as other "ugly" styles.

Providing Content

A programmer can also improve the readability of code by adding information beyond the simple rearrangement of text. Common ways to provide this information are a thoughtful choice of names for constants and variables, and the insertion of useful comments at appropriate locations in the code.

Magic numbers: A magic number is a literal constant (e.g. 7.5) in source code without a name. Unless the meaning of the constant is documented, a maintenance programmer will have a difficult time modifying the code whenever the value of the constant must be changed. One suggested programming style is that all constants should have a meaningful name, unless the value is 0 or 1.

Variable names: Unlike constants, the value of a variable changes during the execution of a program. To provide a way to store and refer to the current value, a variable must be given a name. Ideally, the name will describe what attribute is represented by the variable. Very short names and heavily abbreviated names can be cryptic to the reader.

In many programming languages, the variable name must be declared, a data type specified, and (recommended) an initial value assigned before the variable is used in computations.

Comments: Comments can be placed in source code for most programming languages. Usually, some special marking is required (e.g. "//") to indicate that the comment is not to be executed. Style guidelines from McConnell (2004) state that comments should be included only if they (1) describe the code's intent, (2) provide information not in the code, or (3) summarize a section of code. Java allows several types of comments: full-line comments, end-line comments, and multiple-line comments.

Why Teach Programming Style?

In *The Practice of Programming*, Kernighan and Pike (1999) discuss why we should "bother" with programming style.

"Why worry about style? Who cares what a program looks like if it works? Doesn't it take too much time to make it look pretty? Aren't the rules arbitrary anyway?"

Some of Kernighan and Pike's answers to the above questions include:
(1) "Sloppy code is bad code." Well-written code has fewer errors, and will often be smaller.
(2) "Good style should be a matter of habit." A programmer's work relies on habits developed

ISSN: 2167-1435
v30 n2537

over time. With good habits, it takes much less time to write working programs.

As we pointed out in the Introduction section, customers want programs that are correct, perform well, make efficient use of resources, and are maintainable. Good programming style makes source code more readable and understandable, which helps programmers provide the above features in the software they create. We teach programming style because it helps students develop the ability to write professional quality code.

3. UGLYCODE SOFTWARE

To assist instructors in teaching programming style, we have written a Java program called UglyCode. The UglyCode software presents programming style concepts in "reverse". The usual "forward" approach presented in textbooks shows examples of bad code, and then applies good style principles to improve the code. Many "PrettyPrint" programs are available to demonstrate the forward approach.

For UglyCode, the input is a short Java program that has been written to illustrate good programming style. Using UglyCode, choices can be made on how to "degrade" the style of the code. Students can see how much harder it is to understand source code when good style features are omitted (e.g. indenting).

Used together, instructors can demonstrate programming style concepts with a blend of "forward" (textbook/PrettyPrint) and "reverse" (UglyCode) examples.

UglyCode Program Features
Our explanation of how to use the UglyCode software is in terms of the controls that appear on the main screen (see Appendix B). The controls include a File menu choice, six sets of checkboxes to select style features to change, and two buttons to activate code changes.

File menu
This is the only main menu choice on the UglyCode screen. It includes the following submenu options.

1. *Open*: Open an existing source code file, using a "file-chooser" input control. UglyCode is designed for Java programs, but most features also apply to similar programming languages such as C and C++.

2. *Save As*: As style selections (as described below) are made and implemented, the resulting "ugly" versions of the original program can be saved as text files. To avoid overwriting the input file, the name for each saved file should differ from the input file name.

3. *Exit*: This option ends the UglyCode program.

Checkboxes
Checkboxes are grouped by style category. Within each group, the checkboxes act like command buttons, in that at most one box can be checked. The following checkbox groups are listed on the right-hand side of the UglyCode main screen.

1. ***Line Spacing***: Choices in this group show how the inclusion or exclusion of blank lines in code can affect readability. We include three line spacing options.

Remove Blank Lines: Selecting this checkbox causes all blank lines to be removed from the code. This choice is equivalent to single-spacing, which is a common format for business reports.

Double Space Code: Double-spacing rarely appears in production code, although a few developers embrace it. This style choice is included here to contrast with single-spacing. Students are often asked to use double-spacing when writing term papers for non-computer courses. The extra blank lines in source code are not intended for grading, but can be used for inserting notes during code reviews.

Random Blank Lines: Blank lines can make it easier to see which parts of the code belong together. Unfortunately, blank lines do not improve program readability if the lines are inserted at inappropriate places. This style option randomly inserts blank lines into the code. After each non-blank line, the probability is 1/3 that the next line will be blank, independent of actions on previous lines. The resulting code almost always makes the program logic less clear.

2. ***Indenting***: We provide three options for the number of spaces that occur on the left side of each line of code.

Remove Indents: With this option, all spaces on the left side of each line are trimmed off. All

code starts at the left margin. This makes it difficult to identify where branching and looping control structures start and end.

Add Fixed Size Indents: The "best" size for indenting can be a doctrinal preference among programmers. With this option, the instructor can demonstrate the readability of code with various indenting choices--such as 2 vs. 3 vs. 4 spaces. Zero spaces is equivalent to removing all indents. A pop-up window allows the user to enter the desired number of spaces per indent. Note that more than one indent can appear on a line, such as in nested loops.

Add Random Indents: In this selection, each line receives a random indent size of 0 to 16 spaces. This is clearly not a practical way to indent, but a similar result can occur in practice (and in student assignments).

Suppose that code is written using an editor with a fixed-indent size (say 4), but the programmer mixes spaces with *tabs*. If the code is later brought into a different editor (e.g. Notepad with tab size 8), the mixture of new tab sizes and old spaces can yield a ragged left margin for the code. Debugging ragged-edge code can be a very frustrating experience.

3. ***Curly Braces***: Where to put the curly braces to designate the start and end of blocks is a layout decision guided by language traditions, as well as by programmer preferences. C, C++, and Java have separate histories, with different preferred block marking rules. This option allows the instructor to compare the traditional C-style braces with Java-style braces, and allows students to form their own preferences.

Change to Java Style: With this choice, curly braces that define blocks for loops (while, for) and branches (if-else) are formatted to have the opening brace on the *same* line as the decision expression. Curly braces in other parts of the code are not changed.

Change to C Style: With this choice, curly braces are formatted to have each opening brace on its own *separate* line. Closing braces in the code are not changed.

4. ***Comments***: Comments can improve a programmer's understanding of the *intent* of the code, but only if the comments are "helpful". Options are given to show the negative effects of (1) having no comments, and (2) having "useless" comments. UglyCode only acts on single-line comments that start with "//". Other comment delimiters (e.g. /* and */) are ignored.

Remove Comments: With this option, all comments starting with "//" are removed from the source code. For full-line comments, the entire line is removed. For an end-line comment, only the comment is removed. The source code before the comment remains.

Change to Useless Comments: This option replaces all full-line and end-line comments with "useless" comments. The number of possible useless comments is endless. For full-line comments, UglyCode chooses randomly from a list of 29 computer-humor statements found on the web. For example, one of our favorite full-line comments is: "True Klingon programmers never put comments in their code."

The space for end-line comments is usually shorter, so UglyCode chooses randomly from a list of 11 popular desserts (to tempt a hungry programmer). For example, one end-line dessert is: "Strawberry Shortcake". In either case, the comments are not relevant to the code. Each repeated choice of this option will give a new sample of useless comments.

5. ***Variable Names***: Many variable naming styles are prevalent. Some are language specific (e.g. the preference for lower case names in C). Most naming conventions recommend the use of "meaningful names", subject to possible name length restrictions. For example, "countyTax" is self-descriptive, while “CT” could be misconstrued as an eastern US state or a medical diagnostic procedure.

The variable naming options in UglyCode show how a departure from typical naming rules can affect program understanding.

Change Case: This is a simple option to demonstrate how case differences relate to readability. For all variables with common type declarations (int, byte, char, long, float, double, boolean, and String) appearing at the *start* of a line, the case of each *letter* in the variable name is changed from lower-case to upper-case, or from upper-case to lower-case. For example, a lower-case name such as "job_cost" will be changed to "JOB_COST". A camel-case name such as "netIncome" will become "NETiNCOME". This option allows students to see how annoying minor name changes can be to a programmer.

ISSN: 2167-1435
v30 n2537

Use Meaningless Names: Whether a variable name is considered "meaningless" depends on the context. There are many ways to create meaningless names. We chose a well-known encryption algorithm, a Caesar cipher, because it is easy to program and scrambles text. Each letter in a name is considered case-sensitive, and is changed to the letter three positions later in the alphabet (with wrap-around). Non-letters are unchanged. For example, the variable "bestBUY" would become "ehvwEXB", which looks pretty meaningless.

6. ***Line Breaks***: For free-format languages (e.g. Java), the programmer can choose how much of each statement to place on a line. For short statements, more than one statement can appear on a single line. For long statements, the placement of a line break can affect the readability of the code.

Set Line Length: This option shows what the code will look like if a line length is specified. A pop-up window allows the user to enter a desired minimum line length (e.g. 40). The code is then reformatted so that when concatenated statements exceed this length, they are split over additional lines.

A line break is placed in the first "safe" position at or beyond the minimum length. "Safe" is defined to be immediately after the first semicolon (";"), left brace ("{"), right brace ("}"), or plus-sign ("+ ") at or beyond the minimum length. These break points are not certain to be safe, since they could split statements in "bad" places.

Remove Line Breaks: This is the ultimate reformatting of the source code. All line breaks are removed and replaced with spaces. The program now consists of a single long line. The UglyCode window shows this line without word-wrap, so the window's bottom slider control must be used to view the entire program. The revised one-line program can be saved, and then viewed in an editor that provides word-wrap (e.g. Notepad).

Note: After making programming style changes, the resulting program can be saved as a text file. If you are fortunate, the revised program will compile and run. Before trying to compile, make sure that the class name in the saved Java program matches the output file name.

For example, if Remove Blank Lines, Remove Indents, Remove Comments, Change Case (for Variable Names), and Remove Line Breaks are all checked, the reformatted program (which consists of a single long line) should execute exactly as before the changes. The source code is much less readable, but the computer doesn't mind.

Be aware that Set Line Length and Remove Line Breaks, together with code that includes comments, causes compiling problems when comments are split over two lines or appear between two statements within a line.

Command Buttons
The two buttons on the lower right-hand side of the screen are used to invoke actions on the source code: to make style changes, or to restore the initial code.

Ugly It!: After several style options have been selected using the checkboxes, this button should be clicked to activate the changes on the original source code. The restyled "ugly" code will then appear in the main screen window.

Reset Text: Clicking this button will restore the code in the window to its original form.

Code style changes are not cumulative. Each set of selected changes is applied to the original source code. If an instructor wants to demonstrate the cumulative effects of style changes, she/he should plan a sequence of changes, and then mark cumulative sets of checkboxes for the Ugly It! button clicks.

4. UGLYCODE TEACHING EXAMPLES

Several examples of how style concepts can hinder the clarity of a program are described below. Each example uses sections of the sample Java code listed in Appendix A. This Java program counts the number of prime integers less than or equal to N and compares this count with N/ln(N), which is the asymptotic value stated in the Prime Number Theorem (Newman, 1980). We display just enough UglyCode input and output to demonstrate the effect of the indicated programming style choices.

Blank Lines and Comments
The first source code example demonstrates the combined effect of removing all blank lines and comments. The code section to be transformed

ISSN: 2167-1435
v30 n2537

is presented below. This code includes two full-line comments, two end-line comments, and one blank line.

```
public static boolean isPrime(long X)
{
  // Determine if X is a prime
  if(X < 2) return false;
  if(X == 2) return true;
  if(X % 2 == 0) return false;

  // Check odd integers above 2
  for(long k = 3; k*k <= X; k+=2){
    if(X % k == 0) {
      return false;
    }
  } // end for
  return true;
} // end isPrime
```

Figure 1A: Initial Code.

The code with the comments and blank lines removed is listed next. Fifteen lines have been reduced to twelve lines.

```
public static boolean isPrime(long X)
{
  if(X < 2) return false;
  if(X == 2) return true;
  if(X % 2 == 0) return false;
  for(long k = 3; k*k <= X; k+=2){
    if(X % k == 0) {
      return false;
    }
  }
  return true;
}
```

Figure 1B: Ugly Code With Blank Lines and Comments Removed.

The function name is descriptive and suggests the purpose of the function. The code for the function is still readable, but more mental effort is required to understand the algorithm.

Indenting and Curly Braces

The next code example shows how readability suffers when all indenting is removed, and C-style curly braces are used. The code section to be transformed is displayed below. These seven lines of code include two levels of indenting, one for the block of statements within the for loop, and another for the computation inside the if-statement.

```
// Count actual number of primes
numPrimes = 0;
for(long k = 2; k <= N; k++) {
  if(isPrime(k)) {
    numPrimes++;
  }
} // end for
```

Figure 2A: Initial Code.

The code without indenting but with C-style braces becomes the nine lines listed below. Again, the code is readable, especially with the comments. However, the boundaries of the nested blocks for the if-statement and the for-statement are more difficult to distinguish.

```
// Count actual number of primes
numPrimes = 0;
for(long k = 2; k <= N; k++)
{
if(isPrime(k))
{
numPrimes++;
}
} // end for
```

Figure 2B: Ugly Code With C-style Braces Without Indenting.

C-style curly braces alone will generate more lines of code, since each opening brace is on a separate line. C-style braces with no indenting places all opening and closing braces (often interspersed) along the left margin of the code. This can result in a zigzag pattern of braces.

Useless Comments

The third code sample includes a full-line comment and an end-line comment. The full-line comment describes an input action performed by a section of code. The end-line comment makes it easy to determine where a method (in this case, *main*) ends.

```
public static void main(String args[])
{
  long N, numPrimes;
  double estPrimes, PNTratio;

  // Get maximum N from command line
  N = 1;
  if(args.length >= 1) {
    N = Long.parseLong(args[0]);
  }
  . . .
} // end main
```

Figure 3A: Initial Code.

ISSN: 2167-1435
v30 n2537

For comments to be effective, they must provide information that is helpful to programmers who later read the source code. Here, instead of removing the comments, we replace them with comments that are unrelated to program logic.

```
public static void main(String args[])
{
  long N, numPrimes;
  double estPrimes, PNTratio;

  // 640K ought to be enough for anybody.
  N = 1;
  if(args.length >= 1) {
    N = Long.parseLong(args[0]);
  }
  . . .
} // Chocolate Mousse
```

Figure 3B: Ugly Code With Useless Comments.

The full-line comment is replaced with the historically short-sighted quote "640K ought to be enough for anybody", which has been anecdotally attributed to Bill Gates. The comment that marks the end of the main method now recommends an enjoyable dessert, "Chocolate Mousse".

Meaningless Variable Names

The final source code example demonstrates how variable names can affect program readability. In this example, two long integers and two double floating point numbers are declared. The declared names aren't perfect, but they do suggest what the variables represent, without being unnecessarily long.

```
long N, numPrimes;
double estPrimes, PNTratio;
. . .
// Calculate estimated number of primes
if(N < 2) {
  estPrimes = 0.0;
  PNTratio = 0.0;
} else {
  estPrimes = (double) N/Math.log(N);
  PNTratio = numPrimes / estPrimes;
}
```

Figure 4A: Initial Code.

Using the UglyCode program, we scramble the names of variables that are declared at the start of a statement. (We do not scramble the names of variables declared within statements, such as counter variables declared inside loops.)

UglyCode uses a simple variable renaming algorithm that leads to meaningless names. The algorithm is a Caesar cipher, with a forward shift of 3, applied separately to lower-case and upper-case letters. Since all variables are declared before they are used, we are able to change all variable names consistently with one pass through the code.

The result of using the Caesar cipher on the initial code sample is shown below.

```
long Q, qxpSulphv;
double hvwSulphv, SQWudwlr;
. . .
// Calculate estimated number of primes
if(Q < 2) {
  hvwSulphv = 0.0;
  SQWudwlr = 0.0;
} else {
  hvwSulphv = (double) Q/Math.log(Q);
  SQWudwlr = qxpSulphv / hvwSulphv;
}
```

Figure 4B: Ugly Code With Meaningless Variable Names.

Variable name N is advanced to Q, and numPrimes becomes qxpSulphv. Observe that qxpSulphv has nothing to do with Sulphur. Without the helpful comment, the variable names do not reveal what the code is intended to accomplish.

The above programming style examples illustrate the utility of the UglyCode software for interactive classroom and laboratory use. With six style groups and 2-3 choices per group, there are a total of 14 possible cases involving a single style change. When several style changes are combined into one example, or when the cumulative effects of a sequence of changes are examined, the number of cases increases dramatically.

Of course, not all cases will be of equal interest for a given sample Java program. A variety of sample programs can be prepared to demonstrate specific style concepts and combinations.

5. SUMMARY AND CONCLUSIONS

In this paper, we described ways in which programming style can affect how source code is

ISSN: 2167-1435
v30 n2537

read and understood. We related programming style to more noted program features, such as correctness, performance, efficiency, modularity, and maintainability.

We introduced a program we have written called UglyCode, which allows an instructor to demonstrate how style changes effect the readability of code. Instead of showing an example of "bad" code and then making it "pretty", UglyCode works in the opposite direction. UglyCode input should be a Java program written in "good" style. Style changes are then requested, and the resulting degradation of the code can be viewed immediately.

The UglyCode software allows students to see the effects of individual style changes, as well as groups of changes. A cumulative sequence of style changes can be performed easily. Any transformed source code can be saved in a text file. Students can then attempt to compile and run the modified code to determine whether or not the changes effect how the program runs. It is informative to see how often style changes are ignored by the computer.

Future Research

We have tested early prototypes of the UglyCode software in Programming and Software Engineering courses. The data we have collected from students is largely anecdotal. With a completed version of UglyCode now available, we plan to measure how well this tool helps teach students the importance of good programming style.

Note: An executable version of the UglyCode program, along with the sample Java program, can be obtained from the authors.

6. ACKNOWLEDGEMENTS

An initial version of the UglyCode software was prepared by Samuel Grissom and David Fu, CS students at Azusa Pacific University.

7. REFERENCES

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford (2009). *Introduction to Algorithms* (3rd ed). MIT Press.

Dijkstra, E. W. (1971). A short introduction to the art of programming. *E. W. Dijkstra Archive*. www.cs.utexas.edu/~EWD/

Kernighan, Brian W., & Pike, Rob (1999). *The Practice of Programming*. Addison-Wesley.

Kernighan, Brian W., & Plauger, P.J. (1978). *The Elements of Programming Style* (2nd ed). McGraw-Hill.

Kernigan, Brian W., & Ritchie, Dennis M. (1988). *The C Programming Language* (2nd ed). Prentice Hall.

Klein, Gerwin (2009). Correct OS kernel? Proof? Done! *USENIX ;login:*, 34(6):28-34, Dec 2009.

Klein, G., Elphinstone, K., Heiser, G., et. al (2009). Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA. ACM.

Lafore, Robert (2003). *Data Structures and Algorithms in Java* (2nd ed). Sams Publishing.

Lewis, John, & Loftus, William (2011). *Java Software Solutions, Foundations of Program Design* (7th ed). Addison Wesley.

Liang, Y. Daniel (2012). *Introduction to Java Programming* (9th ed). Prentice Hall.

McConnell, Steve. (2004). *Code Complete* (2nd ed). Microsoft Press.

Newman, D. J. (1980). Simple analytic proof of the prime number theorem. *American Mathematical Monthly* 87.

Sedgewick, Robert, & Wayne, Kevin (2011). *Algorithms* (4th ed). Addison-Wesley.

Shustek, Len (2009). Donald Knuth: a life's work interrupted. *Communications of the ACM*, Volume 51, No 8.

Silberschatz, Abraham, Galvin, Peter B., & Gagne, Greg (2012). *Operating System Concepts* (9th ed). Wiley.

Somerville, Ian (2011). *Software Engineering* (9th ed). Addison-Wesley.

ISSN: 2167-1435
v30 n2537

APPENDIX A: Sample Java Program

```
// Prime Number Theorem
// pi(N) ~ N/ln(N) for large N

public class TestPNT6
{
  public static void main(String args[])
  {
    long N, numPrimes;
    double estPrimes, PNTratio;

    // Get maximum N from command line
    N = 1;
    if(args.length >= 1) {
      N = Long.parseLong(args[0]);
    }

    // Count actual number of primes
    numPrimes = 0;
    for(long k = 2; k <= N; k++) {
      if(isPrime(k)) {
        numPrimes++;
      }
    } // end for

    // Calculate estimated number of primes
    if(N < 2) {
      estPrimes = 0.0;
      PNTratio = 0.0;
    } else {
      estPrimes = (double) N/Math.log(N);
      PNTratio = numPrimes / estPrimes;
    }

    // Output results
    System.out.println("\n Prime Number Theorem"
      + "\n max N.... = " + N
      + "\n primes... = " + numPrimes
      + "\n N/ln_N... = " + estPrimes
      + "\n ratio.... = " + PNTratio);
    return;
  } // end main
```

ISSN: 2167-1435
v30 n2537

```
  public static boolean isPrime(long X)
  {
    // Determine if X is a prime
    if(X < 2) return false;
    if(X == 2) return true;
    if(X % 2 == 0) return false;

    // Check odd integers above 2
    for(long k = 3; k*k <= X; k+=2){
      if(X % k == 0) {
        return false;
      }
    } // end for
    return true;
  } // end isPrime

} // end class
```

APPENDIX B: UglyCode Software Main Screen