DOI:10.1145/2043174.2043191

Article development led by acmqueue
queue.acm.org

What separates good code from great code?

BY ROBERT GREEN AND HENRY LEDGARD

Coding Guidelines: Finding the Art in the Science

COMPUTER SCIENCE IS both a science and an art. Its scientific aspects range from the theory of computation and algorithmic studies to code design and program architecture. Yet, when it comes time for implementation, there is a combination of artistic flare, nuanced style, and technical prowess that separates good code from great code.

Like art, code is simultaneously subjective and non-subjective. The non-subjective aspects of coding include "hard" ideas that must be followed to create good code: design patterns, project structures, the use of common libraries, and so on. Although these concepts lay the foundation for developing high-quality, maintainable code, it is the nuances of a programmer's technique and tools—alignment, naming, use of white space, use of context, syntax highlighting, and IDE choice—that truly make code clear, maintainable, and understandable, while also giving code the ability to communicate intent, function, and usage clearly.

This separation between good and great code occurs because every person has an affinity for his or her own particular coding style based on his or her own good (or bad) habits and preferences. Anyone can write code within a design pattern or using certain "hard" techniques, but it takes a great programmer to fill in the details of the code in way that is clear, concise, and understandable. This is important because just as every person may draw a unique meaning or experience from a single piece of artwork,

every developer or reader of code may infer different meanings from the code depending on naming and other conventions, despite the architecture and design of the code.

From another angle, programming may also be seen as a form of "encryption." In various ways the programmer devises a solution to a problem and then encrypts the solution in terms of a program and its support files. Months or years later, when a change is called for, a new programmer must decrypt the solution. This is usually not an enviable task, which can mainly be blamed on a failure of clear communication during the initial "encryption" of the project. Decrypting information is simple when the necessary key is present. So, too, is understanding old code when special attention has been paid to what the code itself communicates.

To address this issue, some works have defined a single coding standard for an entire programming language,[7] while others have acquiesced to accepting naming conventions as long as they are consistent.[6] Beautiful code has been defined in general terms as readable, focused, testable, and elegant.[1] The more extreme case is the invention of an entire programming language built around a concrete set of ideals, such as Ruby or Python. Ruby emphasizes brevity, simplicity, flexibility, and balance.[4] The principles behind Python are clear in the *Zen of Python*,[5] where the focus lies on beauty, simplicity, readability, and reliability.

Our approach to this issue has been to develop a system of coding guidelines (available online[3]). While these guidelines come from an educational environment, they are designed to be useful to practitioners as well. The guidelines are based on a few broad principles that capture some fundamental principles of communication and elevate the notion of coding conventions to a higher level. The use of these conventions will also improve the sustainability of a code base. This article looks at these underlying principles.

One area not considered here is the use of syntax highlighting or IDEs. While either one may make code more readable (because of syntax highlighting or code folding, among others) and easier to manage (for example, quickly looking up or refactoring functions and/or variables), our guidelines have been developed to be IDE and color neutral. They are meant to reflect foundational principles that are important when writing code in any setting. Also, while IDEs can help improve readability and understanding in some ways, the features found in these tools are not standard (consider the different features found in Visual Studio, Eclipse, and VIM, for example). Likewise, syntax highlighting varies greatly among environments and may easily be changed to match personal preference. The goal of the following principles is to build a foundation for good programming that is independent of the programming IDE.

Figure 1. Use of vertical alignment to show symmetry.

```
char c1;
c1 = getChoice();
switch(c1){
    case 'q': case 'Q': quit();                 break;
    case 'e': case 'e': enterPerson(content);   break;
    case 'd': case 'd': delPerson(content);     break;
    case 's': case 's': sortByName();           break;
    case 'l': case 'l': showAll();              break;
    case 'f': case 'f': searchByName(content);  break;
    case default:       System.out.printIn("--Invalid Command!!\n");
}
```

Figure 2. Example of cluttered presentation.

```
private JFrame mainFrame = new JFrame("Wind Power Calculator");
private JTextArea windVel = new JTextArea(VEL, 2, TEXT_WIDTH);
private JLabel velTag = new JLabel("Wind Velocity");
private JTextArea sweptArea = new JTextArea(SWEPT_AREA, 2, TEXT_WIDTH);
private JLable sweptAreaTag = new JLabel("Swept Area");
private JTextArea genSize = new JTextArea(GEN_SIZE, 2, TEXT_WIDTH);
private JButton calculatePower = new JButton("Calculate Power");
```

Figure 3. Revision of code in Figure 2 showing tabular structure.

```
private JFrame    mainFrame      = new JFrame    ("Wind Power Calculator");
private JTextArea windVel        = new JTextArea (VEL, 2, TEXT_WIDTH);
private JLabel    velTag         = new JLabel    ("Wind Velocity");
private JTextArea sweptArea      = new JTextArea (SWEPT_AREA, 2, TEXT_WIDTH);
private JLable    sweptAreaTag   = new JLabel    ("Swept Area");
private JTextArea genSize        = new JTextArea (GEN_SIZE, 2, TEXT_WIDTH);
private JButton   calculatePower = new JButton   ("Calculate Power");
```

Consider a Program as a "Table"

In a recent *ACM Queue* article, Poul-Henning Kamp[2] makes the fascinating point that much of the style of programming languages stems from the ASCII character set and typewriter-based terminals. Programming languages make no use of the graphical properties and options of modern devices. While code must be written with the clarity of good English grammar, it is not English text. Instead it is more like math and tables.

This is a far-reaching principle. First, it speaks directly to the use of fonts. Do not use a variable-width (proportional) font for program code, as code is not text. Fixed-width fonts (for example, Courier and Data Gothic) look appealing and allow easy alignment of code. Proportional (variable-width) fonts prevent proper alignment, and even more importantly, do not "look like" code.

While one should continue to think of a program as a sequence of actions

or as an algorithm at a high level, each section of code should also be thought of as a presentation of a chart, table, or menu. In figures 1, 2, and 3 notice the use of vertical alignment to show symmetry. This is a powerful method of communication.

In the case when a long line of code spills into multiple lines, we suggest breaking and realigning the code. For example, instead of

```
participant newEntry = new participant
(id, name, address1, address2, city,
state, zip, phone, email);
```

use

```
participant newEntry = new participant
(id, name, address1, address2,
     city, state, zip, phone, email);
```

or

```
participant newEntry = new participant
(id, name, address1, address2, city,
               state, zip, phone, email);
```

Let Simple English be Your Guide

A programmer creates a name for something with full knowledge of its use, and often many names make sense when one knows what the name represents. Thus, the programmer has this problem: *creating a name based on a concept*. The true challenge, however, is precisely the opposite: *inferring the concept based on the name*! This is the problem that the program reader has.

Consider the simple name

`sputn`

taken from the common C++ header file `<iostream.h>`. An inexperienced or unfamiliar programmer may suddenly be mentally barraged with a bout of questions such as: Is it an integer? A pointer? An array or a structure? A method or a variable? Does *sp* stand for saved pointer? Is *sput* an operation to be done *n* times? Do you pronounce it *sputn* or *s-putn* or *sput-n* or *s-put-n*?

We advocate basing names on conventional English usage—in particular, simple, informal, abbreviated English usage. Consider the following more specific guidelines:

- Variables and classes should be nouns or noun phrases;
- Class names are like collective nouns;
- Variable names are like proper nouns;
- Procedure names should be verbs or verb phrases;
- Methods used to return a value should be nouns or noun phrases;
- Booleans should be adjectives;
- For compound names, retain conventional English syntax; and
- Try to make names pronounceable.

Some examples of this broad principle are shown in Figure 4.

There is an interesting but small is-

Figure 4. Examples of basing names on conventional English usage.

Variables		Class Names	
Not the Right Noun	**Better**	**Not the Right Noun**	**Better**
Round	Wheel	Accounting	BankAccount
LoopTimes	NumLoops	SetPoint	Point
Valid	InputStatus	NodeNetworking	SocketInfo
Starting	Source		
Ending	Destination		
Rows	NumRows		

Problematic	Preferable
Person personInfo;	PersonInfo P1, P2;
Socket socketDesc;	SocketDescription socket;
Frame TopFrameSection;	Frame TopFrame;
Message = EmergencyAlertLabels[i]	AlertText = EmergencyLabel[i]

Not the Right Verb	More Readable
NameSet	SetName
Modified	Modify
Withdrawal	Withdraw
Right	MoveRight

Incorrect Function Names	More Readable
numFiles = countFiles(directory);	numFiles = fileCount(directory);
A = computeArea(parcel);	A = Area(parcel);
x = getImagePos(i).x;	x = Image(i).xCoord;

Incorrect Boolean Vars	Grammatically Better
Fill	Full
Terminate	Terminated
Real	isReal
Edit	IsEditable
Waits	Waiting
License	hasLicense

Grammatically Incorrect	Better
IdVehicle	VehicleID
NoSectors	NumSectors
FormEnable();	EnableForm();

Unpronouncable	Pronouncable
Tbl	Table
GenYmDhMs	GenerateTime
Cntr	Counter
Nbr	Num

sue when considering examples such as:

```
numFiles = countFiles(directory);
```

While `countFiles` is a good name, it is not an optimal name since it is a verb. Verbs should be reserved for procedure calls that have an effect on variables. For functions that have no side effects on variables, use a noun or noun phrase. One does not usually say

```
Y = computeSine(X);
or
milesDriven =
computeDistance(location1, loca-
tion2);
```

but rather

```
Y = sine(X);
or
milesDriven = Distance(location1,
location2);
```

We suggest that

```
numFiles = fileCount(directory);
```

is a slight improvement. More importantly, this enforces the general rule that verbs denote procedures, and nouns or adjectives denote functions.

Rely on context to simplify code

All other things being equal, shorter programs are always better. As an example, local variables that are used as index variables may be named `i`, `j`, `k`, and so on. An array index used on every line of a loop need not be named any more elaborately than `i`. Using `index` or `elementNumber` obscures the details of the computation through excessive description. A variable that is rarely used may deserve a long name: for example, `MaxPhysicalAddr`. When variable names are long, especially if there are many of them, it quickly becomes difficult to see what's going on. A variable name can often be shortened by relying on the context in which it is used. For example, the variable `Store` in a stack implementation rather than `StackStore`.

Major variables (objects) that are used frequently should be especially short, as seen in the examples in Figure 5. For major variables that are used throughout the program, a single letter may encourage program clarity.

Use White Space to Show Structure

While written and spoken communication may reach a high level of clarity, it is often left wanting of meaning if not accompanied by the personal touch of nonverbal cues and tendencies. An individual's body language helps clarify the spoken word. In a similar sense, the programmer relies on white space—what is not said directly—in the code to communicate logic, intent, and understanding.

An example is the use of blank lines between conceptually different sections of code. Blank lines should improve readability as they separate logically different segments of the code and thus provide the literary equivalent of a section break. Appropriate places to use blank lines include:

- When changing from preprocessor directives to code;
- Around class and structure declarations;

Figure 5. Keeping names short and simple.

Too Lengthy	Better
LoopIndex	i, j
NumberOfTimes	N (or n)
CheckIfEntryIsCorrect	Validate
IsARealNumber	IsReal
Temporary	Temp

Too Verbose	Preferable
Stack CurrentStack	Stack S
Window Window1, Window2	Window W1, W2
Frame TopFrame	Frame Top
Counter Cntr	Counter C
SearchTree Tree	SearchTree T

Acceptable	Preferable
TreeNode	Node
CustomerID	ID
StackStore	Store
CarDriver	Driver
NameStringInfo	NameInfo

Figure 6. Example of code that uses white space well.

```
public class SimpleAccount {
    private double balance;

    public double  getBalance()          { return balance;}
    public void    setBalance(double b)  { balance = b;}
    public void    deposit(double num)   { balance = balance + num;}
    public void    withdraw(double num)  { balance = balance - num;}

    public static void main(String args[]){
        SimpleAccount my_account;

        my_account = new SimpleAccount();
        my_account.deposit(250);
        System.out.println("Current balance " + my_account.getBalance());
        my_account.withdraw(80.00);
        my_account.withdraw(60.00);
        System.out.println("Remaining balance " + my_account.getBalance());
    }
}
```

▸ Around a function definition of some length;

▸ Around a group of logically connected statements of some length; and

▸ Between declarations and the executable statements that follow.

Consider the code listing in Figure 6. Individual blank spaces should also be used to show the logical structure within a single statement. Strategic blank spaces within a line simplify the parsing done by the human reader. At a minimum, blank spaces should be included after the commas in argument lists and around the assignment operator "=" and the redirection operators "<<" and ">>".

On the other hand, blank spaces should not be used for unary operators such as unary minus (-), address of (&), indirection (*), member access (.), increment (++), and decrement (--).

Also, if it makes sense, put two to three statements on one line. This practice has the effect of simplifying the code, but it must be used with discretion and only where it is sensible to do so.

Let Decision Structures Speak for Themselves

The case statement used in Figure 1 brings up a general point: very simple decision statement structures can be tersely presented, showing the alternative code simply, and, if possible, without braces, as in the example in Figure 7.

It is not uncommon for simple conditions to be mutually exclusive, creating a kind of generalized case statement. This, as is common practice, can be printed as a chain, as in Figure 8.

Of course, it may be that the structures are truly nested, and then one must use either nested spacing or functions to indicate the alternatives. Again, the general point is to let the structure drive the layout, not the syntax of the programming language.

In the brace wars, we do not take a strong stand on the various preferences shown in Figure 9, but we do feel strongly that the indent is vital, as it is the indent that shows the structure.

Focus on the Code, Not the Comments

The ability to communicate clearly is an issue that is faced in all facets of the human experience. Programmers must achieve a level of clarity, continuity, and beauty when writing code. This means focusing on the code and its clarity, balance, and symmetry, not on its length or comments. While this concept does not advocate the removal of comments or negate their use and importance in appropriate situations, it does suggest that programmers must use comments wisely and judiciously. The focus should be on developing code that, for the most part, clearly communicates intent and functionality. This practice will automatically reduce the need for many comments.

Discussion

Although the guidelines presented here are used in an educational setting, they also have merit in industrial environments. Students who are educated using these guidelines will most likely use them (or some variant) as they enter industry. To demonstrate this, we have developed an example that applies these guidelines to two very different styles. The first is the Unix style. It is terse, often making use of vowel deletion, and is often found

Figure 7. Decision statement structure, tersely presented.

```
if(Card != null)   display.setText(Card.getText());
else               display.setText("No More Cards.");
```

Figure 8. Case statement presented as a chain.

```
if (result >= 90)
    cout << "Grade of A!";
else if (result >= 80)
    cout << "Grade of B";
else if (result( >= 70)
    cout << "Sorry, grade of C";
else
    cout << "Not very good";
```

Figure 9. Examples of K&R, ANSI, and Whitesmiths coding styles.

```
if (expression) {
    statements
}
```

```
if (expression)
{
    statements
}
```

```
if (expression)
    {
    statements
    }
```

Figure 10. Example of a systems-programming coding style.

```
//Unix Style
void tokenizeStr(string str, vector<string>& result, const string& delim = " "){
    int pos = 0;
    string strtok;
    for(;;){
        pos = str.find(delim);
        if(pos == (int)string::npos){
            result.push_back(str);
            break;}
        strtok = str.substr(0, pos);
        result.push_back(strtok);
        str = str.substr(pos+1);
    }
}
```

in realistic applications such as operating-system code. This is not to imply that all or most system programmers use this style, only that it is not unusual. Figure 10 shows a small example of this style.

We call the second style the textbook style, as illustrated in Figure 11. Again, this in no way means to imply that all or most textbooks use this style, only that the style in the example is not unusual. In this style the focus is on learning. This means that there is frequent commenting, and the code is well spread out. For the purposes of learning and understanding the details of a language, this style can be excellent. From a practical perspective or for any program of some scale, this style does not work well as it can be overwhelming to use or to read. Moreover, this style makes it difficult to see the overall design, as if one is stuck under the trees and cannot see the forest around.

Figure 11. Example of a textbook coding style.

```
// TEXTBOOK STYLE
void tokenizeString(string myString, vector<string>& listOfTokens,
        const string& tokenDelimiter = " ")
{
    // Precondition: myString is not null
    //
    // Parses myString into a list of tokens using the given delimiter.
    // If no specific delimiter is given, uses the space as a delimiter
    //
    // Postcondition: listOfTokens contains the individual tokens as values

    int index = 0;
    string nextToken;
    boolean loop = true;

    // Obtain tokens and store in vector
    while(loop)
    {
        index = myString.find(delimiter);
        if(index == (int)string::npos)
        {
            // end of string found
            tokenList.push_back(myString);
            loop = false;
        }
        else
        {
            // Append nextToken to vector
            nextToken = myString.substr(0, index);
            tokenList.push_back(nextToken);
            myString = myString.substr(index + 1);
        }
    }
}
```

Figure 12. Example of a coding style using the guidelines presented here.

```
// OUR STYLE
void tokenizeString(string S, vector<string>& tokenList,
    const string& delimiter = " ") {
    // Given a string S, compute the list of its tokens.

    int    position;
    string token;
    boolean moreTokens;

    moreTokens = true;
    while(moreTokens){
        position = S.find(delimiter);
        if(position == (int)string::npos){
            tokenList.push_back(S);
            moreTokens = false;
        }else{
            token = S.substr(0, position);
            tokenList.push_back(token);
            S = S.substr(position + 1);
        }
    }
}
```

Figure 12 is a rework of the function in figures 10 and 11, using the guidelines discussed here to make a smooth transition between academic and practical code. This figure shows a balance of both styles, relying more directly on the code itself to communicate intent and functionality clearly. Compared with the textbook style, the resultant code is shorter and more compact while still clearly communicating meaning, intent, and functionality. When compared with the Unix style, the code is slightly longer, but the meaning, intent, and functionality are clearer than the original code.

Figure 13 illustrates the guidelines presented here in another setting. This is a function taken from a complex program (10,000 lines) related to power-system reliability and energy use regarding PHEVs (plug-in hybrid electric vehicles). The program makes numerous calculations related to the effect that such vehicles will have on the current power grid and the effect on generation and transmission systems. This program attempts to evaluate the reliability of power systems by developing a model for reliability evaluation using a Monte Carlo simulation.

While the previous examples show the merit of the guidelines presented here, one argument against such guidelines is that making changes to keep a certain coding style intact is time consuming, particularly when a version-control system is used. In the face of a time-sensitive project or a project that most likely will not be updated or maintained in the future, the effort may not be worthwhile. Typical cases include class projects, a Ph.D. thesis, or a temporary application.

Figure 13. Realistic and complex example of code following the guidelines presented here.

```
void loadDataFile(double& pLoad,      double& qLoad,
          int&    numBuses, int&     numTransLines, string systemName,
          vector<Generator>& gens, vector<Line>& transLines, vector<Bus>& buses){
   // This function loads the various system parameters from the power system data file.
   // The power system data is encoded as a csv file,

    ifstream       systemData;
    string         dataLine;
    vector<string> dataItem;
    int            numGens;

    systemData.open(("../Data/" + systemName).c_str());
    if (systemData.is_open()) {
        systemData >> numGens;
        systemData >> pLoad;
        systemData >> qLoad;
        systemData >> numBuses;
        systemData >> numTransLines;

        numGens = 0;

        //Clear Vectors
        gens.clear(); transLines.clear(); buses.clear();

        // Set Generators
        for(int i = 0; i<numGens; i++){
            systemData >> dataLine;
            Utils::tokenizeString(dataLine, dataItem,",");

            gens.push_back(Generator(
                    atof(dataItem[3].c_str()), atof(dataItem[4].c_str()),
                    atof(dataItem[5].c_str()), atof(dataItem[6].c_str()),
                    atof(dataItem[7].c_str()), atoi(dataItem[0].c_str()))
                );

            gens[i].setIndex(i);
            dataItem.clear();
        }

        // Set transmission lines
        for(int i = 0; i<numTransLines; i++){
            systemData >> dataLine;
            Utils::tokenizeString(dataLine, dataItem,",");

            transLines.push_back(Line(
                atoi(dataItem[0].c_str()),  atoi(dataItem[1].c_str()),
                atoi(dataItem[2].c_str()),  atof(dataItem[3].c_str()),
                atof(dataItem[4].c_str()),  atof(dataItem[5].c_str()),
                atof(dataItem[6].c_str()),  atof(dataItem[7].c_str()),
                atof(dataItem[8].c_str()),  atof(dataItem[9].c_str()),
                atof(dataItem[10].c_str()), atof(dataItem[11].c_str()),
                atof(dataItem[12].c_str()), atof(dataItem[13].c_str()))
            );
            dataItem.clear();
        }

        // Set bus loadings
        for(int i=0; i<numBuses; i++){
            systemData >> dataLine;
            Utils::tokenizeString(dataLine, dataItem,",");
            buses.push_back(Bus(
                atoi(dataItem[0].c_str()),  atoi(dataItem[1].c_str()),
                atoi(dataItem[6].c_str()),  atoi(dataItem[10].c_str()),
                atof(dataItem[2].c_str()),  atof(dataItem[3].c_str()),
                atof(dataItem[4].c_str()),  atof(dataItem[5].c_str()),
                atof(dataItem[6].c_str()),  atof(dataItem[7].c_str()),
                atof(dataItem[12].c_str()), atof(dataItem[11].c_str()),
                atof(dataItem[9].c_str()))
            );
            dataItem.clear();
        }
        systemData.close();
    }
}
```

If, however, the codebase in question has a long lifespan or will be updated and maintained by others (for example, an operating system, server, interactive Web site, or other useful application), then almost any changes to improve readability are important, and the time should be taken to ensure the readability and maintainability of the code. This should be a matter of pride, as well as an essential function of one's job. C

Related articles on queue.acm.org

Beautiful Code Exists, if You Know Where to Look
George Neville-Neil
http://queue.acm.org/detail.cfm?id=1454458

Software Development with Code Maps
Robert DeLine, Gina Venolia, and Kael Rowan
http://queue.acm.org/detail.cfm?id=1831329

Reading, Writing, and Code
Diomidis Spinellis
http://queue.acm.org/detail.cfm?id=957782

References

1. Heusser, M. Beautiful code. *Dr. Dobb's* (Aug. 2005); http://www.ddj.com/184407802.
2. Kamp, P-H. Sir, please step away from the ASR-33! *ACM Queue 8*, 10 (2010); http://queue.acm.org/detail.cfm?id=1871406.
3. Ledgard, H. Professional coding guidelines. 2011 Unpublished report, University of Toledo; http://www.eng.utoledo.edu/eecs/faculty_web/hledgard/softe/upload/.
4. Molina, M. What makes code beautiful. *Ruby Hoedown*, 2007.
5. Peters, T. *The Zen of Python.* PEP (Python Enhancement Proposals). Aug. 20, 2004; http://www.python.org/dev/peps/pep-0020/.
6. Reed, D. Sometimes style really does matter. *J. Computing Sciences in Colleges 25*, 5 (2010), 180-187.
7. Sun Developer Network. Code conventions for the Java programming language, 1999; http://java.sun.com/docs/codeconv/.

Acknowledgments
The authors would like to thank David Marcus and Poul-Henning Kemp for their insightful comments while completing this work, as well as the software engineering students who have contributed to these guidelines over the years.

Robert Green is pursuing his Ph.D. at the University of Toledo. He has multiple years of experience developing software across a variety of industries. His research interests include biologically inspired computing, high-performance computing, and alternative energy.

Henry Ledgard was a member of the design team that created the programming language ADA, a language he believes was a creative, sound design. He is the author of several books on programming, and is a professor at the University of Toledo. His research interests include principles of language design, human engineering and effective ways to teach CS.