**Good Code, Bad Code: Toward Production Quality Code**

*Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live.*

--Martin Golding

### Code quality

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is understandability. This is because in any non-trivial team project, team members expect their code to be read, understood, and modified by other developers later on. This handout interprets ‘good quality code’ primarily as code that is easy for developers to work with.

There are many coding practices you can follow to improve code quality. For example, the two code samples given below achieve the same functionality and follow similar coding standards, but one is of better quality than the other (Which one? Why?).

```c
// Good example
int subsidy() {
    int subsidy;
    if (!age) { // not over age limit
        if (!sub) { // not subsidized
            if (!notFullTime) { // not part time
                subsidy = 500;
            } else {
                subsidy = 250;
            }
        } else { // already subsidized
            subsidy = 250; // overAgeLimit
        }
    } else { // overAgeLimit
        subsidy = -1; // already subsidized
    }
    return subsidy;
}
```

```c
// Bad example
int calculateSubsidy() {
    int subsidy;
    if (isSenior) { // reject senior
        subsidy = REJECT_SENIOR;
    } else if (isAlreadySubsidized) { // subsidized
        subsidy = SUBSIDIZED_SUBSIDY;
    } else if (isPartTime) { // fulltime subsidized
        subsidy = FULLTIME_SUBSIDY*RATIO;
    } else { // not subsidized
        subsidy = FULLTIME_SUBSIDY;
    }
    return subsidy;
}
```

![Figure 9. Code comparison: same function, different quality](image)

Here are some simple things you can do to improve your code (adapted from [2], Chapter 11).

#### Follow a standard

A coding standard is specific to a programming language and specifies guidelines such as the location of opening and closing braces, indentation styles and naming styles (e.g. whether to use Hungarian style, Pascal casing, Camel casing, etc.). It is important that the whole team/company use the same coding standard and that standard is commonly used in the industry. Doing so will make it easier for others to work with your code, be they members from the same team or transferred from another team/company. IDEs can help to enforce some parts of a coding standard e.g. indentation rules.

#### Name well

Proper naming improves the code quality immensely. It also reduces bugs caused by ambiguities regarding the intent of a variable or method. Here are some things to consider:

- Related things should be named similarly, while unrelated things should NOT.
- Use nouns for classes/variables and verbs for methods.
  For example, LimitChecker is a better choice for a class name than CheckLimit
A name should be an accurate and complete description of its entity. For example, processStuff() is a bad choice for a method; what 'stuff' and what 'process'? On the other hand, removeWhiteSpaceFromInput() is a better choice. Other bad examples include flag, temp, i (unless used as a conventional control variable for a loop).

It is preferable not to have lengthy names, unless unavoidable; names that are 'too short' are even worse.

Use correct spelling in names (and even in comments). Avoid 'texting-style' spelling.

If you must abbreviate or use acronyms, do it consistently. Explain their full meaning at an obvious location.

Avoid misleading names (e.g. those with multiple meanings), similar sounding names, hard to read ones (e.g. avoid ambiguities like "is that a lowercase l, capital L or number 1?", or "is that number 0 or letter O?"), almost similar names, foreign language names, slang, and names that are only meaningful within specific contexts such as private jokes.

Minimize using numbers or case to distinguish names (value1 vs value2, Value vs value).

Distinguish clearly between single valued and multivalued variables (e.g. Person student, ArrayList<Person> students).

Be obvious

We can improve code understandability by making things explicit, even if the language syntax allows them to be implicit. Here are some examples:

- Use explicit type conversion instead of implicit type conversion.
- Use parentheses/braces to show grouping even when they can be skipped.
- Use enumerations when a certain variable can take only a small number of finite values. For example, instead of declaring the variable 'state' as an integer and using values 0,1,2 to denote the states 'starting', 'enabled', and 'disabled' respectively, declare 'state' as type SystemState and define an enumeration SystemState that has values 'STARTING', 'ENABLED', and 'DISABLED'.
- When statements should follow a particular order, try to make it obvious (with appropriate naming, or at least comments). For example, if you name two functions 'taskA()' and 'taskB()', it is not obvious which one should be called first. Contrast this with naming the functions 'phaseOne()' and 'phaseTwo()' instead. This is especially important when statements in one function must be called before the other one.

No misuse of syntax

It is safer to use language constructs in the way they are meant to be used, even if the language allows shortcuts. Here are some examples:

- Use the 'default' option of a case statement for an intended default action and not just to execute the last option. If there is no default action, you can use the 'default' branch to detect errors (i.e. if execution reached the 'default' branch, throw an exception). This also applies to the final 'else' of an if-else construct. That is, the final 'else' should mean 'everything else', and not the final option. Do not use 'else' when an 'if' condition can be explicitly specified, unless there is absolutely no other possibility.
<table>
<thead>
<tr>
<th>Not recommended</th>
<th>Better</th>
</tr>
</thead>
</table>
| if (red) print "red";
  else print "blue"; |
| if (red) print "red";
  else if (blue) print "blue";
  else error("incorrect input"); |

- Use one variable for one purpose. Do not reuse a variable for a different purpose other than its intended one, just because the data type is the same.
- Do not reuse formal parameters as local variables inside the method.

```
Not recommended
double computeRectangleArea(double length, double width) {
    length = length * width;
    return length;
}

Better
double computeRectangleArea(double length, double width) {
    double area;
    area = length * width;
    return area;
}
```

- Avoid data flow anomalies such as using a variable before initialization, or pre-assigning values to variables and modifying it without prior use of the pre-assigned value.

### Avoid error-prone practices

Some coding practices are notorious as sources of bugs, but nevertheless common. Know them and avoid them. Here are some examples of how to avoid such error-prone coding practices:

- Never write a case statement without the 'default' branch.
- Never write an empty 'catch' statement.
- All 'opened' resources must be closed explicitly. If it was opened inside a 'try' block, it should be closed inside the 'finally' block.
- If several methods have similar parameters, use the same parameter ordering.
- Do not include unused parameters in the method signature.
- Whenever an arithmetic calculation is performed, check for overflow and 'divide by zero' errors.

### Minimize global variables

Global variables may be the most convenient way to pass information around, but they do create implicit links between code segments that use the global variable. Avoid them as much as possible.

### Avoid magic numbers

Avoid indiscriminate use of numbers as constants (e.g. 3.1415 as the mathematical constant PI) all over the code. Define them as named constants, preferably in a central location. A similar logic applies to string literals with special meanings (e.g. "Error 1432").

```
[Bad]   return 3.14236;
[Better] return PI;
```
Along the same line, make calculation logic explicit rather than using the final value.

[Bad]    return 9;
[Better] return MAX_SIZE - 1;

Imagine going to the doctor's and saying "My eye1 is swollen"! Minimize the use of numbers to distinguish between related entities such as variables, methods and components.

[Bad]  value1, value2
[Better] originalValue, finalValue

**Throw out garbage**

We all feel reluctant to delete code we have painstakingly written, even if we have no use for that code any more ("I spent a lot of time writing that code; what if we need it again?"). Consider all code as baggage you have to carry; get rid of unused code the moment it becomes redundant. If you need that code again, simply recover it from the revision control tool you are using (you are using one, aren't you?). Deleting code you wrote previously is a sign that you are improving.

**Minimize duplication**

The book "The Pragmatic Programmer" calls this the DRY (Don't Repeat Yourself) principle. Code duplication, especially when you copy-paste-modify code, often indicates a poor quality implementation. While it may not be possible to have zero duplication, always think twice before duplicating code; most often there is a better alternative.

**Comment minimally, but sufficiently**

*Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?’ Improve the code and then document it to make it even clearer.* – Steve McConnell

Some think commenting heavily increases the 'code quality'. This is not so. Avoid writing comments to explain bad code. Improve the code to make it self-explanatory.

Do not use comments to repeat the obvious. If the parameter name clearly indicates its purpose, refrain from repeating the description in a comment just for the sake of 'good documentation'. Write the occasional comment so as to help someone understand the code more easily.

Do not write comments as if they are private notes to self. Instead, write them well enough to be understood by another programmer. One type of comments that is almost always useful is the header comment that you write for a file, class, or an operation to explain its purpose.

When you write comments, use them to explain 'why' and 'what' aspect of the code, rather than the 'how' aspect. The former is usually not apparent in the code and writing it down adds value to the code. The latter should already be apparent from the code.

*The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.* – Edsger Dijkstra

**Be simple**

Often, simple code runs faster. In addition, simple code is less error-prone and more maintainable. Do not readily dismiss the brute-force yet simple solution for a complicated one,
unless your application justifies the need for high-performance requirements (e.g. an application processing huge amount of data or providing concurrent access to thousands of users). As the old adage goes, "keep it simple, stupid" (KISS). Do not try to write clever code.

*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*

--Brian W. Kernighan

*Programs must be written for people to read, and only incidentally for machines to execute.*

--Abelson and Sussman

*Premature optimization is the root of all evil in programming.*

--Donald Knuth

**Code for humans**

Always assume that "anyone who reads the code you write is dumber than you (duh)". Make your code understandable to a "dumb" person. The smarter you think you are compared to your teammates, the more effort you need to make your code understandable to them. Even if we do not intend to pass the code to someone else, code quality is still important because we all become 'strangers' to our own code someday. Here are some tips:

- Avoid long methods. Be wary when a method is longer than the computer screen, and take corrective action when it goes beyond 50 LOC (lines of code). The bigger the haystack, the harder it is to find a needle.
- Limit the depth of nesting. According to the Linux 1.3.53 CodingStyle documentation, "if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program".

In particular, avoid *arrowhead* style code.

- Limit to one statement per line.
- Avoid complicated expressions, especially those having many negations and nested parentheses. Sure, the computer can deal with complicated expression; but humans cannot. If you must evaluate complicated expressions, have it done in steps (i.e. calculate some intermediate values first and use them to calculate the final value).
- The default path of execution (i.e. the path taken when everything goes well) should be clear and prominent in your code. It is the ‘unusual’ cases that should be inside if blocks. Someone reading the code should not get distracted by alternative paths taken when error conditions happen. One technique that could help in this regard is the use of guard clauses [http://tinyurl.com/guardclause].

<table>
<thead>
<tr>
<th>Not Good</th>
<th>Better</th>
</tr>
</thead>
</table>
| if (!isUnusualCase) { 
  if (!isErrorCase) { 
    start();
    process();
    cleanup();
    exit();
  } else { 
    handleError();
  } 
} else { 
  handleUnusualCase();
} | Guard clause
if (isUnusualCase) { 
  handleUnusualCase();
  return;
} Guard clause
if (isErrorCase) { 
  handleError();
  return;
} start();
process();
cleanup();
exit();
• Alternative conditions are separated from their handling.
• Main path is nested deeply.
• Alternative paths dealt with first so that the reader doesn’t have to remember them for long.
• The main path is un-indented.

Tell a good story
Lay out the code so that it adheres to the logical structure. The code should read like a story. Just like we use section breaks, chapters and paragraphs to organize a story, use classes, methods, indentation and line spacing in your code to group related segments of the code. For example, you can use blank lines to group related statements together.

Sometimes, the correctness of your code does not depend on the order in which you perform certain intermediary steps. Nevertheless, this order may affect the clarity of the story you are trying to tell. Choose the order that makes the story most readable.

A sure way to ruin a good story is to describe details which are irrelevant to the story. Avoid varying the level of abstraction within a code fragment. Note: The Productive Programmer (by Neal Ford) calls this the SLAP principle i.e. Single Level of Abstraction Per method.

<table>
<thead>
<tr>
<th>Bad</th>
<th>Better</th>
</tr>
</thead>
</table>
| readData();
salary = basic*rise+1000;
tax = (taxable?salary*0.07:0);
displayResult(); | readData();
processData();
displayResult();

SIDE NOTE

Abstraction
Most programs are written to solve complex problems involving large amounts of intricate details. It is impossible to deal with all these details at the same time. The guiding principle of abstraction stipulates that we capture only details that are relevant to the current perspective or the task at hand. For example, within a certain software component, we might deal with a ‘user’ data type, while ignoring the details contained in the user data item. These
details have been 'abstracted away' as they do not affect the task of that software component. This is called data abstraction. On the other hand, control abstraction abstracts away details of the actual control flow to focus on tasks at a simplified level. For example, print("Hello") is an abstraction of the actual output mechanism within the computer.

Abstraction can be applied repeatedly to obtain higher and higher levels of abstractions. For example, a File is a data item that is at a higher level than an array and an array is at a higher level than a bit. Similarly, execute(Game) is at a higher level than print(Char) which is at a higher than an Assembly language instruction MOV.

Do not release temporary code
We all get into situations where we need to write some code momentarily that will be disposed of or replaced with better code later. Such code should be marked in some obvious way (e.g. using an embarrassing print message) so that you will not forget to remove them later. It is irresponsible to retain such code in a product release. Production code should be worthy of the quality standards you live by. If a code is important enough to be released, it is important enough to be of production quality.

Programs should be written and polished until they acquire publication quality. --Niklaus Wirth

Keep looking for ways to improve code quality
There are many more good coding practices that are not listed above. If you are serious about improving your code quality, try to look out for more such books. Clean code (by Robert C. Martin) and Code complete (by Steve McConnell) are particularly good resources.

FAQ

- Q: What is the difference between good code and clean code?
  A: Depends on how you define 'good' (i.e., good for what?). Besides, 'clean' can be defined in different ways too. The first chapter of the book 'Clean Code' is about how different prominent practitioners define what 'clean' code is. In CS2103, good code is code that is easy to understand for other programmers, and clean code is easier to understand than messy code. For us, good code should be clean and more.

References