[Handout for L2P1]
Good Code, Bad Code

Code quality

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is understandability. That 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, two code samples given below achieves the same functionality and follows similar coding standards, but one is of better quality than the other.

```
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 {
             subsidy = 250;  //overAgeLimit
          }
    }else {
           subsidy = -1; // already subsidized
    }
     return subsidy;
}
```

```
int calculateSubsidy()  {
     int subsidy;
     if (isSenior) {
        subsidy = REJECT_SENIOR;

     } else if (isAlreadySubsidized) {
        subsidy = SUBSIDIZED_SUBSIDY ;

     }else if (isPartTime)
        subsidy = FULLTIME_SUBSIDY*RATIO ;

     }else {
        subsidy = FULLTIME_SUBSIDY;
     }
      return subsidy;
  }
```

Figure 2. Code comparison: same function, different quality

Here are some simple things you can do to improve your code (adapted from [1], Chapter 11).

Follow a standard

A coding standard is specific to a programming language and specifies things such as where to place 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 to others to work with your code whether they are other members of the same team or those moved to your team from another team/company.

Name well

Proper naming improves the code quality immensely. It also reduces bugs caused by misunderstandings about what a variable/method does. Here are some things to consider:

- Related things should be named similarly, while unrelated things should NOT be named similarly.
- Use nouns for classes/variables and verbs for methods.

- A name should completely and accurately describe what it names. For example, processStuff() is a bad choice; what 'stuff'? what 'process'?. removeWhiteSpaceFromInput() is a better choice. Other bad examples include flag, temp, i (unless used as the control variable of a loop).
- It is preferable not to have 'too long' names, but it is OK if unavoidable. 'Too short' names are worse.
- Use correct spelling in names (and even in comments). Avoid texting-style spelling.
- If you must abbreviate or use acronyms, do it consistently, and explain their full meaning in an easy-to-find location.
- Avoid misleading names (e.g. those with double meaning), similar sounding names, hard to read ones (e.g. avoid having to ask "is that simple l or capital I or number 1?", or "is that number 0 or letter O?"), almost similar names, foreign language names, slang, and names that can only be explained using private jokes.
- Minimise 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 understandability of the code by explicitly specifying certain things 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 to show grouping even when you can skip them.
- Use enumerations when a certain variable can take only a small number of finite values. For example, instead of declaring the variable 'status' as an integer and using values 0,1,2 to denote statuses 'starting', 'enabled', and 'disabled' respectively, declare 'status' as type SYSTEM_STATUS and define an enumeration called SYSTEM_STATUS 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 'phaseOne()' and 'phaseTwo()', it is obvious which one should be called first.

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 a real default and not simply 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', not the final option. Do not use 'else' to catch an option that can be specified more specifically, unless that is the absolutely the only other possibility.

Not recommended	if (red) print "red"; else print "blue";

Better	if (red) print "red";
	else if (blue) print "blue";
	else error("incorrect input");

- Use one variable for one purpose. i.e. do not reuse a variable for a different purpose than for what it was declared just because the data type is the same.
- Do not reuse parameters received as local variables inside the method.
- Avoid data flow anomalies such as variable being used before initialization and assigning values to variables but changing it later without using the 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 order.
- Do not include unused parameters in the method signature.
- Whenever you do an arithmetic calculation, check for 'divide by zero' problems and overflows.

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 with special meanings (e.g. 3.1415 to mean the value of mathematical ratio PI) all over the code. Define them as 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 vein, make calculation logic behind a number explicit rather than giving the final result.

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

Imagine going to the doctor's and saying "My nipple1 is swollen"! Minimise 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 wrote ourselves, even if we do not use 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. Should you eventually need that code again, you can simply recover it from the revision control tool you are using (you are using one, are you not?). Deleting code you wrote previously is a sign that you are improving.

Minimize duplication

The Pragmatic Programmer book 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 is not possible to have zero duplication, always think twice before duplicating code; most often there is a better alternative.

Make comments unnecessary

> *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 students 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 what is already obvious from the code. If the parameter name already clearly indicates what it is, there is no need to repeat the same thing in a comment just for the sake of writing 'well documented' code. However, you might have to write comments occasionally to help someone to help understand the code more easily. Do not write such comments as if they are private notes to self. Instead, write them well enough to be understandable to another programmer. One type of code 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 dismiss the brute force yet simple solution too soon and jump into a complicated solution for the sake of performance, unless your application has very high performance requirements (e.g., an application processing huge amount of data or providing concurrent access to thousands of users) and you have proof that the latter

solution has a sufficient performance edge. As the old adage goes, KISS (keep it simple, stupid - don't 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

In particular, always assume anyone who reads the code you write is dumber than you (duh). This means you need to make the code understandable to such a person. The smarter you think you are compared to your teammates, the more effort you need to make your code understandable to them. What 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 after we spend some time away from it. Here are some tips that can help you there:

- Avoid long methods. Be wary when a method is longer than the computer screen, and take corrective action when it goes beyond 200LOC. The bigger the haystack, the harder it is to find a needle.
- Limit the depth of nesting. (how many levels is too many? This is what Linux 1.3.53 CodingStyle documentation says: *If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program*.)
- Have only 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, do it 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 if everything goes well) should be clear and prominent in your code. 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].

Tell a good story

Lay out the code to follow the logical structure of the code. The code should read like a story. Just like we use section breaks, chapters and paragraphs to organise 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 tell details not relevant to the story. Do not vary the level of abstraction too much within a piece of code.

[Bad]
```
readData();
salary = basic*rise+1000;
tax = (taxable?salary*0.07:0);
displayResult();
```

[Better]
```
readData();
processData();
displayResult();
```

SIDE NOTE

Abstraction

Most problems our programs try to solve are complex and involve large amounts of intricate details. It is impossible for us to deal with all those details at the same time. Abstraction is our main weapon against such 'overload' of details. Abstraction tells us to capture only those details about something that are relevant to the current perspective or the task at hand. For example, from within a certain software component we might deal with a data type called 'user' without bothering about what details contained in a user data item. We say that those details have been 'abstracted away' because they do not concern the task of that component. This is called *data abstraction*. Another type of abstraction is called *control abstraction*. That is, we abstract away details of the actual control flow to focus on a simpler view of what is happening. For example, we might say print("Hello") or a=a+1 which are in fact much simpler views of what is actually happening in the computer.

The process of abstracting can be applied repeatedly to arrive at higher and higher abstractions. This is what we call *levels of abstraction*. For example, a File is data item that is higher level than an array and an array is higher level than a bit. Similarly, execute(Game) is higher level than print(Char) which is higher than an Assembly language instruction MOV.

Do not release temporary code

We all get into situations where we need to write some code 'for the time being' that we hope to dispose of or replace with better code later. Mark all such code in some obvious way (e.g. using an embarrassing print message) so that you do not forget about their temporary nature later. It is irresponsible to release such code for others' use. Whatever you release to others should be worthy of the quality standards you live by. If the code is important enough to release, it is important enough to be of production quality.

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

Sign your work

To quote from The pragmatic programmer (by Andrew Hunt and David Thomas)**Error! Reference source not found.**, *We want to see pride of ownership. "I wrote this, and I stand behind my work."* By making you put your name against the code you wrote (e.g. as a comment at the beginning of the code), we encourage you to write code that you are proud to call your own work.

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

References

[1] *Practical Tips for Software-Intensive Student Projects* (3e), by Damith C. Rajapakse, Online at http://StudentProjectGuide.info

---End of Document---