Writing Clean Code

Group 9: Lukas Escher, Cole Franz, Ashkan Hemmati, Scott McKay, James Robertson, Andrew Tang, Stanislav Tserkovny

Contents

Introduction

The topic of clean code is a very ambiguous topic that has hundreds of different answers; many of which are accurate. Writing clean code is not doing just *one* thing perfectly and thus there is no simple way to describe clean code due to many variables which will vary on a code to code basis. The definition of clean code in general means is code that can be easily comprehended, tested, maintained, and reused by others in the future. These things all complement each other in one way or another, and they're dependant on each other although there is no clear starting or ending points to writing code. Code that is easy to comprehend will be easier to test, code that is easy to test is easy to maintain, code that is easier to maintain is easier to reuse in the future, and so forth.

On top of the qualities of clean code we must discuss why clean code is important. Clean code is important on a couple main accounts: modifiability and extensibility. These characteristics are specifically good in a world where people don't know what they want and requirements change constantly. Clean code is also important as disorganized and sloppy code often leads to mistakes resulting in memory leaks or other unwanted, unforeseen system failures. Finally, writing clean code is key for developers in how easy it is to find bottlenecks within the system and other restraints that may slow down the development process.

A good analogy for the importance of clean code could be the construction and maintenance of a house. If you want to build a house with a good base and room for expansion you don't just get out your hammer and 2x4s and start building, you need to set a blueprint that you follow to a point to create a perfect house elsewise you end up with boards everywhere and layers being supported or reliant on things that it shouldn't need to stand. The rest of the document will help with setting up and maintaining code as you would a house.

Characteristics of Clean Code

I. Readability (Learnability)

Readability is an important part of clean code it allows for others to follow and learn your code. There are various ways to contribute to the readability of code; documentation, naming variables and methods, and using logical patterns.

Documentation is a way to give the developer a way to follow the code through comments and description in areas throughout the code. Documentation affects readability by giving information that cannot be shown in the code, allows giving a descriptive insight to how a method works, and can give a general oversight of the whole code or a section.

Naming variables and methods in a descriptive way is a huge help when trying to read code. Naming methods is a good way to convey what the main goal of that section of code is going to do. Naming variables allows for understanding just exactly what the variables role is in the given method or in the general code itself. These traits are an important part to clean code, they makes the code almost human readable which makes it a lot easier to learn/understand the code.

Logical patterns are a general reusable solution to a commonly occurring problem within a given context in software design. Patterns make the code more readable because they are easier to follow then random coding style they can be predicted where it is headed and how it may get there.

II. Documentation

When creating software, the code by itself is insufficient; there must be descriptions of its various features and intended operations. Today's software development requires that several people be able to modify codes concurrently. For this reason, it is imperative that others are able to quickly understand and modify other's code. There is nothing more time consuming than having to trace someone else's code or figuring out the innards of the system.

Documentation can be both beneficial and problematic; it increases understanding of complex large-scale systems, reduces maintenance time and allows for effective problem corrections and system improvements. However, poor or outdated documentation can lead to confusion and can diminish progress. We will explore characteristics of both good and bad documentation, as well as various techniques to avoid producing lousy documentations.

Characteristics of Good Documentation

Many styles of documentation are meant to describe the functionality of the code. These types of comments include specifying the pre-conditions and post-conditions of complex functions or modules and/or giving details about specific lines in the code. Furthermore, the parameters inputted are described in more detail if not obvious.

Other documentations report warnings of potential consequences that may arise from doing specific operations. These types of documentations are intended to notify the developer about expected behaviour of the program under given circumstances. Documenting these parts in the code - if they exist - will help avoid producing similar mistakes later on in the coding stage or even at the testing stage when sections of the code must be modified.

In addition to these types of comments, sections that are commented as "TODO" are designed to remind the developers about sections of the code that were skipped or functionality that have not yet been implemented. Although this type of comment is trivial and more of a personal preference, it makes the job easier later on and thus is considered a good thing to keep in mind.

Characteristics of Bad Documentation

Although there are many examples of good documentation approaches, there are many more styles that are considered bad forms of documenting. These documentations are usually caused by inaccurate or wrong ways of executing some of the good style of documentations described above.

Long and unnecessary comments are categorized as bad styles of documentation because they describe the code in so much detail that the comment is no longer useful. In this

case, the comment becomes so confusing that the cost of tracing the code would be less than the cost of understanding the comment itself. This causes a similar result as not writing any comments in the first place. Rule of thumb is to always give the least amount of comment in order to get your point across.

Misleading documentations are produced when the person, other than the coder, misunderstands the intent and writes documentation based on false reasoning. However, there are ways to avoid this type of documentation error; the developers must write the documentations themselves as they write the code. At the same time, this is not always feasible as there are instances where developers must modify other's code and as a result some of the documentations. Therefore, the best way to avoid writing misleading comments is to do more research on vague sections or, as a last resort, leave out the comment altogether.

In other situations, the reason behind misleading documentations is outdated comments that are created as the code is modified. There are things that can be done to deal with outdated documentations such as not documenting simple and small areas of code that are self-explainable, or documenting parts of the code that are less likely to be changed later on.

III. Cohesion and Coupling

III.I Cohesion

Cohesion is the concept of putting things together that belongs together. This concept particularly applies in object- oriented (OO) design but is not limited to. The main benefits of high cohesion are readable, maintainable, and testable code. Cohesion has various type, the common ones are described in table 1.0 below.

Type	Level	Description
Coincidental	Low	Tasks that are together by coincidence, they have no meaningful relationship to each other
Logical	Low	Tasks that all do the same thing but do not necessarily belong together
Temporal	Low	Tasks that are executed at around the same time
Procedural	Medium	Tasks that are executed in a particular order
Communicational	Medium	Tasks that support the same type of data. This could include accessing and modifying that data
Sequential	Medium	Tasks where the output of one task becomes the input of another task, enforcing a sense of order of those tasks
Functional	High	The "Holy Grail" of cohesion, tasks are grouped together because they finish a larger task

Table 1.0

Cohesion is an important part of a vital OO principal, encapsulation. Through encapsulation, information and functionalities are hidden between modules. If encapsulation is

used properly, cohesion is enforced. For example, if all member variables of a class are private variables then the methods that use and/or modify these variables must be within this class (or a specialization of this class). On the other hand, if member variables are made public, then this would allow for external modules to modify/use these variables, thus lowering cohesion.

III.II Coupling

Coupling is the measure of how dependent different modules are on each other. While aiming for zero coupling is unreasonable (because modules need to be able to communicate), low coupling is desirable in software design. Primary drawbacks of high coupling include low testability, low readability, and low reusability.

Some signs of high coupling within code are a large amount of global variables, public member variables of classes, methods (especially constructors) calling many methods from other classes, and large classes. Global variables are necessary at times but the overuse of them can increase coupling severely. The more modules that use these global variables to communicate with each other, the more difficult it would be to modify them. One change in any module would probably mean the other modules need to be changed and changing anything about the global variable would almost guarantee that changes would be required.

While highly coupled code can function, it makes it almost impossible to reuse and test. If there is a module within a project that is coupled with other modules that could be reused, it becomes extremely difficult to only take the module you want. To reuse a coupled module, all the other modules coupled with it would also have to be extracted lowering the benefit of code reuse. For testing purposes, coupled code is a nightmare. For example, a class where its constructor calls other constructors of different classes (and maybe those call more constructors!) would be infeasible. Situations like this would make unit testing impossible without extremely large set-up and teardown functions. The readability of coupled code tends to be very low. Again consider the previous situation where a constructor calls other constructors. It would be fairly challenging for a new programmer to understand all the different method calls required for just one constructor! To reduce coupling, create modules that require minimal communication with other modules. Some strategies for this would be to use proper encapsulation techniques and careful planning. UML class diagrams can be extremely helpful for this as they clearly show how different classes interact.

IV. Refactoring

Refactoring is the process of rewriting existing code in place without changing its external functionality. This process improves code readability, code maintainability, reduces coupling, reduces code complexity, and generally can improve all aspects of code. Refactoring is tricky and can be very difficult and time consuming. The benefits, however, can greatly outweigh the cost.

Why refactor instead of simply rewriting parts of the application? Refactoring can be minimally intrusive while re-architecting and rebuilding is very intrusive. This means that with refactoring, the code base could be fully functional throughout the process. Also with refactoring, a smaller portion of the code can be targeted while the rest of the module or

subsystem can go untouched. With the less intrusive changes the process can be stopped and started whenever convenient, meaning that the team working on the application is still able to react quickly to changes.

Refactoring is a tedious process, with the utmost care and attention required to avoid changing the behaviour of the application. It is for this reason that it is done in small steps, with thorough testing done after each step to ensure that the application still functions the same as before. Small steps are used to minimize the complexity of the operation. When refactoring, a small piece should be selected, evaluated, and once that process is done, modified. Testing must be done immediately after each step of the process in order to ensure no new bugs have been introduced. A strong unit test suite will be invaluable and will dramatically decrease the amount of effort and the chances of introducing bugs and errors.

Techniques for refactoring are very similar to techniques for writing good code from scratch. Basically the aim is to give code better abstraction, break the code into more logical pieces, to improve compliance to naming conventions, and improve location of code. Some useful techniques are:

1. Encapsulate fields. Provide getters and setters instead of having public variables.
2. Use generalized types. Methods that accept many different types of variables can be re-used more. An example of this is the list classes provided by many languages; one class can handle many variable types.
3. Extract methods and classes. Pulling code into its own method or class allows that code to be used by many other methods which reduces repetition.
4. Good naming conventions. Giving methods, classes, and variables proper names greatly increases code readability and reduces bugs caused by inappropriate naming.

Reading the refactoring techniques above you are probably wondering why refactoring is necessary, why not just write good code in the first place? The reason is that code evolves over time. Perhaps a new feature is requested that uses a lot of the same functionality as an old feature. The old feature will have its functionality close as it may not have been required by other pieces of the software when originally built. In this case you would need to refactor the old modules and extract functionality so that your new modules could make use of the common code.

When is it time to refactor code? The first thing that must be determined is whether the cost of refactoring is less than the cost of just rewriting pieces of the code from scratch. Sometimes there are defined times when refactoring is necessary. Take the example where new features are being added and old code can be re-used, in this scenario you know that refactoring needs to be done as a part of completing the new feature. Mostly however knowing when to refactor is based on intuition. If a section of the application is old and messy, perhaps not following current style standards, then it will be more difficult to maintain. Code that is difficult to maintain is a prime candidate for refactoring. Another candidate for refactoring is code that needs to be optimized. Refactoring will remove duplicated code and improve structure which can help optimize slow sections of code.

V. Testing

Testing is an important, yet often neglected part of software development. The presence

of comprehensive test suites, while adding more work during the initial development phases of the software, generally helps minimise problems in later stages of the software by uncovering problems early, helping stop new problems from being introduced, as well as forcing developers to look at their code from a different perspective, often spurring new understanding of the problem and the system.

In order for testing to be most effective as well as easy to implement, it is best to incorporate it into the initial design. Designing systems to be more easily testable not only saves a significant amount of work incorporating the tests, but also generally produces higher quality code. Producing testable code means that as much as possible, the design should strive to maximise modularity. Modular code allows more aspects of the program to be tested in isolation, providing a much more finely grained pin pointing of the causes when something goes wrong.

Finally, the design should include clear interfaces between program components. The more clearly defined the interfaces are, the easier it is for tests to be written without having to consider the exact implementation details of the individual functions and the more accurately they can reflect the system specifications.

Testing can often be misunderstood and thus examples 1.0 through 1.4 will be used to demonstrate poor tests and how it contributes to unclean code.

```
public Card(String n, Country t) {
        setName(n);
        setCountry(t);
}
```

```
public Country(int p, String id, String n,
      Continent c, int a, int b) {
        <<code>>
}
```

Example 1.0

Some systems are poorly designed for testing. This can stem from several areas of the design. Highly coupled systems introduce an element of difficulty when writing tests. In example 1.0, attempting to test some member functions of the Card class involves not only creating an instance of the Card class but also an instance of Country and Continent. Not only does this increase the complexity of the test with a significant amount of extraneous initialization calls but also can cause a test of a function unrelated to either Countries or Continents to fail, creating confusion about where the actual fault in the code is.

```
public double[] getStatistics(StatType type) {
        double[] statistics = new double[Statistics.size()];
```

```
        for (int c = 0; c < statistics.length; c++) {
                statistics[c] = ((Statistic)
            Statistics.elementAt(c)).get(type);
        }
        return statistics;
}
```

Example 1.1

Documentation is also crucially important, especially when developing tests for an existing codebase. Poor documentation can cost test developers a significant number of hours simply attempting to decipher what a module is trying to do. Example 1.1 is an undocumented function. There is no description of what the inputs and outputs of the function are, no description of what the function aims to do, and both Statistics and StatType are classes included in other source files. Tracking down where the other classes are defined, let alone what they do would take a significant amount of time on the part of the tester, making writing tests extremely difficult.

In cases of particularly unclean, poorly tested code, there may be instances of functions that have inconsistent output. Testing the functionality of a subroutine is particularly tricky when it is not predictable.

```
public Risk() {
        // default Android value does not work
        // 10,000 gives StackOverflowError on android on default map
        // 100,000 gives StackOverflowError on android on the map "The Keep"
        // 1,000,000 gives StackOverflowError on android on the map "The Keep"
        // if you place all the troops
        // 10,000,000 very rarely gives crash
        // 100,000,000 crashes on "Castle in the Sky" map on Android (CURRENT
        // VALUE)
        // 1,000,000,000 still crashes on "Castle in the Sky" (also crashes
        // 32bit java SE)
        // 10,000,000,000 still crashes on "Castle in the Sky" (also crashes
        // 32bit java SE)
        // 100,000,000,000 still crashes on "Castle in the Sky" (also crashes
        // 32bit java SE)
        // 1,000,000,000,000 crashes the whole Android JVM
```

Example 1.2

The above example demonstrates a significant lack of clarity in the way the system is supposed to function. It is the culmination of many smaller defects that interact unpredictably and create unstable software. This is what software testing attempts to avoid.

While well intentioned, writing tests can sometimes get out of hand. Testing without a mind to input equivalence classes can create an inflation of test cases that wastes time and makes the test suite harder to understand.

```
@test
public void testPlayerNameBob()
{
        Player player = new Player(“Bob”);
        assertEquals(player.getName(), “Bob”);
}
@test
public void testPlayerNameSteve()
{
        Player player = new Player(“Steve”);
        assertEquals(player.getName(), “Steve”);
}
```

Example 1.3

In example 1.3, both tests test the same functionality and a fault in the function will either be caught by none of them or by all of them.

Another common pitfall to avoid when writing tests is to include too much functionality in one test.

```
@test
public void testPlayers()
{
        Player player = new Player(“Bob”);
        assertEquals(player.getName(), “Bob”);
        Player player = new Player(“Steve”);
        assertEquals(player.getName(), “Steve”);
}
```

Example 1.4

In the above test example, testPlayers, multiple functions are tested at once. This causes a problem with localising the problem when something in the test goes wrong and it becomes difficult to distinguish which of the functions in the test was actually the problem. Writing quality tests is an important aspect of making sure your testing routines are effective as writing hundreds of tests may be unhelpful if the tests have a significant amount of overlap. The main consideration when writing tests is to maximise test coverage and reduce test redundancy.

Writing test suites is not the only way to test software. A number of other techniques exist that allow for finding defects and assuring quality. Additional approaches including code reviews, exploratory testing, and load testing can round out a test suite and greatly improve the validation process of any software.

Although the practice of testing is frequently underappreciated and underutilized, testing has a myriad of positive benefits on code quality, where finding bugs is only a small part. Incorporating tests into the design generally eliminates a number of poor designs from being even considered. Designing test cases frequently improves the general understanding of the

problem and how the system addresses it, and with a quality suite of automated tests, future modifications to the software becomes significantly easier.

VI. Maintainability

Maintenance is often misinterpreted as simply fixing defects when it actually covers a wide spectrum of tasks that includes isolating bugs, adapting to environmental changes (such as porting to a new hardware or OS), modifying code to meet changing or new requirements, improving performance and adding preventative measures to ensure the software does not operate below unacceptable levels. Having good maintainability in your code can mean the difference between changing a few lines of code and overhauling it entirely.

Naive developers often do not consider the costs of maintaining their software after delivering it to their clients when, in truth, maintenance claims about 40-80% of all project costs. Lehman was one of the first people to address the concern of system evolution and maintenance in 1969 and since then, many techniques and publications were made to deal with this issue. Maintainable code tries to lessen the chance of breaking the system when modifying its components. In the following, we will explore the ways in which to increase code maintainability and minimize time spent on making changes.

Before diving into the techniques, let us first explore issues that maintenance developers may encounter with codes that have bad maintainability.

VI.I Maintainability Issues

i) Limited Understanding of the System

The software developer does not have a clear understanding of the system and has to spend extra time reading the code to gain a better understanding. Up to 40-60% of the developer's time may be spent on just reading the code, depending on the complexity of the system's implementation. A solution might be to ask the designer or system coder for help, but that may not be a reliable source of information because people typically lose understanding of the system over time or may be unavailable for various reasons (i.e. left the company).

ii) Testing Large-Scale Systems

Testing of large-scale systems is usually expensive and very time consuming. This problem is especially big in coupled systems where removing and modifying methods may lead to cascading test case failures. Also, depending on the changes, pre-existing tests may become logically incorrect. For example: changing variable's type and name.

iii) Impact Analysis

Impact analysis refers to the complete analysis of a system modification, including what systems and software products are impacted. This is done before coding, and is important to help isolate system errors that may be introduced by the modification, give an estimation of the costs and resources required, and help show the benefits and ramifications of the change to others.

iv) Difficult to do Cost Analysis and Negotiate With Client

Maintenance cost is not obvious. The software developer needs to have a good understanding of both the system and the impact of implementing changes. Thus, it is difficult to negotiate with the client regarding requirements, deadlines, staffing, resources etc.

VI.II Maintainability Techniques

Now we go over the techniques to avoid the pitfalls listed above.

i) Use Good Programming Practices to Increase Learnability

Having good documentation is a viable way of an increasing system comprehension later down the road. The software developer should try to comment not just how the system works, but also why the system is designed in such fashion. Also, the developer should use proper coding practices such as using descriptive variable and function names, as well as choosing the appropriate coding patterns to meet requirements. These steps will help the maintenance developer gain a better understanding of the system much quicker.

ii) Regression testing

Depending on the system's scale, it may not be viable to run the entire test suite every time a change is made. The developer should instead consider *regression testing*, which is a technique that seeks to uncover new bugs while minimizing the resources required for testing by selecting a minimum set of tests that covers an adequate area of the unchanged domains.

For example, if modifications were made to component A and not B, and B has already been tested to work, developer can use a generic test to see if B returns the correct value and focus making (or updating) test cases for A.

iii) Use Database Tools

Database tools such as bug trackers are invaluable to have, as they provide a searchable form to record information regarding what a modification fixes and breaks, as well as which components were modified. With large scale systems, this is helpful in organizing and storing the impact analyses.

iv) Maintainability Measurement Program

Over the years, algorithms were developed to help indicate the maintainability of a system using lines-of-code measures, McCabe measures and Halstead complexity measures. Most IDE's have a plugin that computes the maintainability index, which can be used to help gauge maintenance costs.

v) Reverse Engineering Legacy Systems & Reengineering Design to be Adaptable

Some systems (legacies in particular) are simply too outdated and require an overhaul. In this case, the developer should reverse engineer the system. This requires looking through the code files in depth, generating diagrams to represent the system's different levels of abstraction (such as conceptual and flow diagrams), and finally reengineer the design. This technique is considered extreme and very expensive.

Conclusion

To conclude, it is apparent that there are many factors involved in producing clean code and there is no quick and easy approach and it will take time and effort to produce in order to prevent spending much more time and effort further down the line. Clean code has been shown to require many key characteristics which we managed to sum up to eight key topics. Each characteristic holds its own weighting on its importance of the production of clean code and often one or multiple of these will be neglected resulting in unclean code which is bad in more ways than one, inverse to the results of having clean code. As discussed, the importance of clean code is generally summarized to modifiability and extensibility, as they extend into most other important features of clean code, as requirements are constantly changing and as long as code is easily read to be extended and tested, the job of the clean code has been successfully fulfilled.

References

1. Basili and Boehm. “Software defect reduction top 10 list,” Computer, vol. 34, no. 1, 2001.

2. Beizer. “Software Testing Techniques”, International Thomson, Computer Press, Boston, 1990.

3. Binder, Robert. “Design for Testability in Object-Oriented Systems”, Communications of the ACM, v.37, 1994.

4. Binder, Robert: “Testing Object-Oriented Systems: Models, Patterns, and Tools”. Addison Wesley,y, 2000
   Digital: Link

5. Fowler, Martin: “Refactoring: Improving the design of existing code”. Addison Wesley, 5th ed., 2006
   Digital: Link

6. Martin, Robert: “Clean Code: A Handbook of Agile Software Craftsmanship”. Prentice Hall, Courier, Stoughton, 2008
   Digital: Link

7. Ebeling, C. E: “An Introduction to Reliability and Maintainability Engineering”. McGraw-Hill Companies, Inc., Boston, 1997.

8. Barker, Thomas: “Writing Software Documentation”, Pearson Education, 2nd ed. Upper Saddle River, 2003

9. Newton, Chris. “Software Maintainenance.” Internet: www.clarityincode.com/software-maintenance/, Jan. 9, 2010 [Mar. 18, 2014].

10. Eberly, Wayne, “CPSC 333: Coupling and Cohesion” http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Design/cohesion.html