CS3318 Advanced Programming with Java
Assignment 1, 8 October 2021

Student name: Jack O'Connor
Student number: 119319446

Clean Code
A Personal Report

—

"It is not enough for code to work." - Robert C. Martin

What is clean code?

Clean code is simply a colloquial term used in the programming world which is used to describe code which follows software development best practices. These best practices exist to create a codebase which is structured such that teams of developers can all work together synchronously and asynchronously on their own and others' code without personal quirks or the laziness of other developers infringing on their ability to progress the project.

The following serve as an overview of my understanding as to what the attributes of good clean code are.

Consistent

Clean code must be readily legible to any programmer regardless of their familiarity with the project. This legibility is achieved through the use of *style guides* which provide a familiar skeleton and shape to all programmes. Consistency in coding style allows programmers to immediately intuit aspects of a programme such as whether expressions are variables, constants, classes, methods etc.

This consistency in code style also allows the practiced programmer to quickly cast their gaze and focus on the most relevant code blocks on which they are working, with minimal effort. This is achieved through the use of formatting techniques such as consistently indented statements and control structure braces.

Maintainable

If a piece of code was required to only ever work once in a very specific context then maintainability would not be a factor once the task had been completed. However, this is not how most code is used. New use cases for applications are always being dreamed up by users and designers, and the compatibility of these applications with modern hardware, middleware and software is in constant need of update. Hence, code must be maintainable.

To be maintainable means to be robust enough to withstand frequent changes. Changes to the codebase must only have local effects and not cause errors in other areas of the codebase due to unforeseen dependencies. To achieve this one must keep in mind concepts such as low coupling and high cohesion and utilising modular design approaches such as APIs.

Coupling is a measure of how interdependent your classes/modules are. You could think of this as being analogous to how many import statements are at the top of your class definition. It is a problem for a class to be dependent on another, as the removal of the latter will cause breaking changes in the former. As such, low coupling is to be desired.

Cohesion on the other hand is a measure of how related the attributes and methods of a class are to each other. If half your string methods are in class A and the other half in class B, any string dependent application will automatically increase its coupling beyond what is necessary as it needs both classes to function, while also incorporating whatever irrelevant methods are included in the incohesive classes A and B.

APIs are great in that they naturally lead to low coupling, high cohesion development as related methods are kept in their relevant APIs and the black box nature of API usage encourages APIs to be readily interchangeable and thus have low coupling.

Self-documenting

There is a massive flaw inherent in all codebase documentation. If it is not meticulously kept up to date with the current state of the codebase it is practically useless, and at great expense too. As you can imagine, this is not great. Enter self-documenting code. Self-documenting code is intrinsic to the codebase. It is always up to date with the codebase since it *is* the codebase.

The key to self-documenting code is abstraction. All code is based on some form of model. Take for example a booking system. For an application to correctly solve all of the tasks required of a booking system, a programmer must have a functioning model of a booking system in their head that they are implementing. The difference is whether or not this model is also explicit in their code.

A single booking class could absolutely solve this problem. But this class would be so convoluted, with so many methods and attributes with seemingly little to no relation, that the specific functionality of any part of that class would be hard to grasp. Much better would be to have multiple classes such as rooms, guests, hosts, etc each with their own explicit functionality evident in their individual method names, parameters and return types - all corresponding to functions defined in the overall conceptual model.

Error-free

It would be myopic to describe any broken code as still being “clean”. To know if a piece of code is error-free it must survive a battery of tests designed to catch any possible bug. Robert Martin’s book Clean Code lays out a set of principles for designing tests called FIRST. These principles are core to Test Driven Development and keep production code to a high, clean standard.

FIRST tests are:

- **Fast:** Developers should not be discouraged from using tests on their code by long wait times.
- **Independent:** Every test should be capable of running by itself without the need for a prerequisite test, speeding up testing of specific errors.
- **Repeatable:** Tests should be independent of running environments, depending only on the code that is being tested. This removes the possibility a test was failed due to external dependencies to the code being tested.

- **Self-validating:** Tests should return either true or false for quick verification of success or failure.
- **Timely:** Tests should be written before the piece of production code they are testing in order to guarantee an adherence to prior specifications and remove the temptation of moving the goalposts.

Ranking best practice guidelines of clean code

The following is my personal ranking of the guidelines outlined in the assignment statement.

1. Using naming conventions
2. Importance of code style (formatting)
3. Minimise the accessibility of classes and members
4. Using exceptions rather than return codes
5. Ordering class members by scopes
6. Minimize mutability
7. Using interface references to collections
8. Using enums or constant classes instead of constant interface
9. Avoid using for loops with indexes
10. Using underscores in numeric literals

Why I picked my Top 3

Using naming conventions

Good naming conventions allow programmers to understand if a piece of code is a class, method or variable from its name alone. Once there is an agreed upon standard: classes being mixed case nouns; methods being mixed case verbs with the first letter lowercase; variables being short yet descriptive and mixed case with the first letter lowercase; then with experience a programmer can quickly grasp the structure of any given piece of code.

The reason I have naming conventions as my number one is because, more so than any other guideline, the misuse of naming conventions does not just make a piece of code harder to read, but actively misleads the comprehension of a piece of code. Imagine the cognitive dissonance that a programmer might experience when looking at an integer declared variable being assigned what looks like a class. Or when iterating through the characters of a string seeing a temporary variable called ‘ltr’ instead of the more conventional ‘c’.

Good naming conventions contribute to the consistency and self-documentation of a codebase.

```
String IP_ADDRESS = "127.0.0.0/8"; // Class, constant
int smallNumber = 1_000; // literal, variable
ThisIsAClass theClass = ThisIsAClass.createClass(); // Class, variable, class method

string ipaddress = "127.0.0.0/8"; // something, something?
int SmallNumber = 1_000; // something literal, something?
this_is_a_class the_class = this_is_a_class.create_class(); // huh?
```

Importance of code style (formatting)

Scope is an incredibly important concept in programming. Good formatting makes the scope of variables obvious. Bad formatting, much like bad naming conventions, actively misleads programmers trying to determine scope. Through good use of indentation and alignment of braces with their respective control statements and method/class declarations one can illustrate scope visually.

Scope is vital to enforcing the crucial principle of encapsulation in object oriented programming. It follows that any clean coding guideline which enforces good scope usage must itself have similar import in enforcing clean coding.

Good formatting contributes to the self-documentation and maintainability of a codebase.

```
for (int x : listOne) {
      do(x);
      for (int y : listTwo) {
            do(x, y);
            for (int z : listThree) {
                  do(x, y, z);
            }
      }
}

for (int x : listOne) {
do(x);
for (int y : listTwo) {
do(x, y);
for (int z : listThree) {
do(x, y, z);
}}}
```

Minimise the accessibility of classes and members

Minimising the accessibility of classes and members is a seemingly pointless exercise at first. After all, if you were to remove all the accessibility modifiers at runtime wouldn't your application run exactly the same?

But taking this viewpoint is very shortsighted. Yes an application can be made with entirely public classes and members, and yes that application can even be made to respect the principles of object oriented programming such as encapsulation, abstraction, polymorphism and inheritance. But actually creating an application that respects all these theoretical principles without having some external tool that tells you every time you are making a mistake, or even more likely that a coworker building on your unfamiliar code is making a mistake, is an impossible task.

Minimising the accessibility of classes and members allows the Java compiler to prevent you from modifying variables or constants after they take their first value. It prevents classes from accessing other classes' attributes that should be encapsulated. It even helps with inheritance by preventing certain final classes from being extended further.

The Java compiler will be far more accurate at enforcing the proper rules for the programmer who takes full advantage of its features.

```
public final class Dog {
      private final int numLegs;

      public final Dog( int numLegs  ) {
            this.numLegs = numLegs;
            this.numLegs = 5;  // <---- Not allowed because numLegs final
      }
}
```

New class

```
public class Cat extends Dog {  // <---- Not allowed because Dog final
class
    Dog dog = new Dog( 4 );
    dog.numLegs = 5;  // <---- Not allowed because numLegs private.
}
```

Why I picked my Bottom 2

Avoid using for loops with indexes

This guideline's purpose is to eliminate errors in for loops where you inadvertently alter the index variable in such a way that you are no longer sequentially accessing each element in a collection eg. accidentally incrementing your index variable thus skipping an element in the collection.

This error is easily avoided with good naming conventions such as using the characters i, j or k for indexes. The programmer then knows to not do any operation that changes the value of these indexes while in the scope of the relevant for loop. Provided you follow higher ranked guidelines, not using indexes provides minimal benefit.

```
for (int i = 0; i < numList.length(); i++) {
      doSomething(numList[i]);
}

// vs

for (int x : numList) {
      doSomething(x);
}
```

Using underscores in numeric literals

Having a necessary, hard-coded number so large that it is a struggle to tell what order of magnitude it is is a very rare situation and this is the only time using underscores in numeric literals would be useful. A pair of glasses would probably serve you better.

```
int x = 1000000000;
int x = 1_000_000_000;
```

References

FIRST principles - Robert C. Martin, "Clean Code"