Criterion and principles of easily-modified program

A Prutzkow1,2

1 Department of Computational and Applied Mathematics, Ryazan State Radio Engineering University named after V. F. Utkin, 59/1 Gagarina str., Ryazan 390005, Russian Federation
2 Department of Mathematics, Physics, and Medical Computer Sciences, Ryazan State Medical University named after I. P. Pavlov, 9 Vysokovoltnaya str., Ryazan 390026, Russian Federation

mail@prutzkow.com

Abstract. Ease of modifying is one of the characteristics of the program. The existing approaches do not clarify this concept. These approaches are clean code and maintainability. We formulate simple and explicit criterion and principles of easily-modified programs. The criterion is an E.W. Dijkstra’s criterion. The principles are (1) the names of the elements should reflect their purpose in the program; (2) use indentation for nested statements and blank lines; (3) one program element must have one assignment and one assignment must correspond to only one element; (4) optimize dependencies between program elements. We introduce a metric for easily-modified programs as well. It’s a number of commands in methods and static blocks in a class.

1. What is a Good Program?

Usually there are many programs to solve a problem. How to choose a good program? To say that one program is better than another, you need to choose a comparison criterion. Programs can be compared according to various characteristics: performance, the amount of required computer resources, the length of its program text, usability, etc.

Changes to the program do not end after the developer transfers it to the client. The less time it takes to make changes to the program, the cheaper it is to maintain. Therefore, an important characteristic of the program is the ease of modifying it.

There are two approaches to clarify this concept. These are clean code and maintainability.

2. Approaches Clearifying Ease of Modifying Program

2.1. Clean Code

Original source of clean code is the R. Martin's book [1]. In the «What is Clean Code?» section he collected some citations of famous programmers about clean code. They are following:

(1) «Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.»
(2) “Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.”

(3) “I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code’s author, and if you try to imagine improvements, you’re led back to where you are, sitting in appreciation of the code someone left for you – code left by someone who cares deeply about the craft.”

(4) “You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.”

Clean code determines the quality of the software [2]:

(5) “There are several reasons why you should care about code quality. First, the quality of your code will determine the quality of the software. If you think you can have great software with messy code, you’re dreaming. Second, clean code saves time. You will spend more time reading your code more than you spend writing it. The other people in your team will also spend a good part of their time reading your code. And you will spend time reading other people’s code too. Clean code is easy to read, so clean code saves time. Third, your code is a reflection of yourself. If you don’t value quality in your code, you won’t get hired. Messy code means you don’t hold yourself, or your work, to a very high standard. Clients and employers demand quality.

Clean code is maintainable. It is easy to read and easy to understand. That means it’s easy to use and easy to change. Clean code is straightforward.”

Citation is from [3]:

(6) “There is no sole or strict definition of clean code. Moreover, there is probably no way of formally measuring clean code, so you cannot run a tool on a repository that could tell you how good, bad, or maintainable or not that code is. Sure, you can run tools such as checkers, linters, static analyzers, and so on. And those tools are of much help. They are necessary, but not sufficient. Clean code is not something a machine or script could tell (so far), but rather something that us, as professionals, can decide.

For decades of using the terms programming languages, we thought that they were languages to communicate our ideas to the machine, so it can run our programs. We were wrong. That’s not the truth, but part of the truth. The real language behind programming languages is to communicate our ideas to other developers.

Here is where the true nature of clean code lies. It depends on other engineers to be able to read and maintain the code. Therefore, we, as professionals, are the only ones who can judge this. Think about it; as developers, we spend much more time reading code than actually writing it. Every time we want to make a change or add a new feature, we first have to read all the surroundings of the code we have to modify or extend. The language (Python), is what we use to communicate among ourselves.”

Citation is from [4]:

(7) “A major misunderstanding is to confuse clean code with something that can be called “beautiful code.” Clean code has no beauty reasons. Professional programmers are not paid for writing beautiful or pretty code. They are hired by development companies as experts to create customer value.

Code is clean if it can be understood and maintained easily by any team member.

Clean code is the basis for being fast. If your code is clean and test coverage is good, a change or a new function just take a few hours or a couple of days – and not weeks or months – until it is implemented, tested, and deployed.
Clean code is the foundation for sustainable software and keeps a software development project running over a long time without accumulating a large amount of technical debt. Developers must actively tend the software and ensure it stays in shape because the code is crucial for survival of a software development organization.

Clean code is also a key to make you a happier developer. It leads to a stress-free life. If your code is clean and you’re feeling comfortable with it, you can keep calm in every situation, even in front of a tight project deadline.

All of the points mentioned above are true, but the key point is this: Clean code saves money! In essence, it’s about economic efficiency. Each year, development organizations lose a lot of money because their code is in bad shape.

2.2. Maintainability Guidelines

«Maintainability (how easily a system can be modified) is one characteristic of software quality. Performance (how slow or fast a system produces its output) is another» [5]. The international standard ISO/IEC 25010:2011 defines maintainability as one of eight software quality characteristics. J. Visser presents the following maintainability guidelines [5]: (1) Write short units of code. (2) Write simple units of code. (3) Write code once. (4) Keep unit interfaces small. (5) Separate concerns in modules. (6) Couple architecture components loosely. (7) Keep architecture components balanced. (8) Keep your codebase small. (9) Automate development pipeline and tests. (10) Write clean code.

Thus, maintainability is more general than clean code. Moreover, the 1st, 2nd, and 3rd guidelines are the clean code rules in [1].

Visser describes an approach for quantifying maintainability that is more practical than the emotional definitions of "clean code" in Martin's book.

Clean code and maintainability are blurred. We define the «easily-modified program» term by four simple and explicit principles.

3. Purpose of the Study

Clean code is a vague («there is no sole or strict definition of clean code») and emotional («reads like well-written prose», «beautiful code», «is also a key to make you a happier developer») term. Read the citations above again and tell what clean code is. The citations don’t make the term clearer.

Maintainability is more general term than clean code. So maintainability inherits vagueness of clean code.

Purpose of the study is to introduce a criterion and explicit principles of easily-modified program, a measure to compare easily-modified program as well.

Modifying of the program consists of two main stages: (1) program reading; (2) properly program modifying. These two stages are closely related, so the principles of writing easily-modified programs apply to both the first and the second stage.

We analyze the citations, own programming experience, and formulate a criterion and principles.

4. Criterion of Easily-Modified Program

The criterion of easily-modified program is «program structure should be such as to anticipate its adaptation and modification» [6].

5. Principles of Easily-Modified Program

5.1. The names of the elements should reflect their purpose in the program

This is the cornerstone of writing easily-modified programs. Rejecting it negates all other principles.

Before changing a program, you need to understand how it works. The time to understand the work can be reduced if the program itself tells about itself, and not using the program documentation.

For the program to talk about itself, you need to name the program elements (for example, variables, routines, classes, methods, etc.), based on their purposes.
5.2. Use indentation for nested statements and blank lines
Correct naming is not the only thing that shortens the time it takes to understand the program text. Equally important is the location of the statements in a line. The indentation of statement nesting indicates the beginning and end of conditional statements, iteration statements, methods, and classes. Blank lines separate fields, constructors, methods and their logically separate parts. Avoiding indents and blank lines increases the time it takes to understand the program.

We don’t mention the «only one statement in the line» rule in this section, because we suppose the rule is accepted.

5.3. One program element must have one assignment and one assignment must correspond to only one element

5.3.1. One class should do one task and one task should be done by one class
All classes can be divided into data classes and utility classes depending on their functional purpose.

The data class describes an entity of the domain. It usually contains fields and methods to get and set them values. Fields are characteristics of the entity and constitute the state of the object. Based on the principle under consideration, a data class should describe one entity, but describe it in its entirety.

The utility class has no non-const fields and only contains methods. Based on the principle under consideration, a utility class should contain methods that perform logically related actions. All other methods should be moved to other utility classes.

5.3.2. One method must do one action and one action must be done by one method
Methods create a new language that raises the level of abstraction. The program can be written in a new language using ready-made methods. If a method performs multiple actions, then it must be split. The method should be considered for splitting if its length exceeds 6-8 lines.

5.3.3. Eliminate duplicate program fragments
Duplication of program fragments is the presence in the program of a fragment that is functionally equivalent to another fragment of the same program.

There are two types of program text duplication [7]: (1) Identical: identical lines in program text. (2) Reducible: program lines can be altered to another one by such as a change of line sequence, renaming of variable, type or routine names, replacing statements by functionally equivalent ones, etc.

Duplication forces, when changing one fragment, to change its duplicate as well, which increases the time of program change.

5.4. Optimize dependencies between software elements

5.4.1. Dependency concept
The program is divided into parts, and these parts interact with each other.

A dependency is a reference in the text of one program element to another element. The program is woven from dependencies. In a class, there may be a mention of another class, for example, when inheriting or when declaring fields. In a method, methods of the same or other classes can be called.

Dependencies lead to the fact that when you change an element, you need to change all its references in other elements.

5.4.2. Optimizing dependencies
The purpose of dependency optimization is to reduce program change time. Optimization is carried out in two directions.

I. Reduction of redundant dependencies. The fewer dependencies in the program, the less time it takes to change the program. It is impossible to make software elements completely independent from each other. But some dependencies may be redundant. Therefore, they should be reduced. An easily
modified program has no redundant dependencies. In design patterns [8] number of dependencies is minimal.

II. Replacing some dependencies with other dependencies so as to reduce the time to change the program. Dependencies have two main characteristics:

1) stability — the ability to remain unchanged when changing the program; the higher the abstraction used in the dependency has, the more stable this dependency is;

2) reliability — the ability to support the performance of specified functions by associated program elements with different input data. For instance, a dependency of string and literal in line 4 is more reliable than a dependency in line 3 (Listing 1), because the dependency in line 4 doesn’t throw a NullPointerException.

Listing 1. Two Approaches to Compare String and Literal

```java
1    String word = "abc";
2
3    boolean isStringEqualToLiteral = word.equals("xyz");
4    boolean isLiteralEqualToString = "xyz".equals(word);
```

5.4.3. Divide programs into levels of abstraction

Program actions can be divided into levels. Level properties:

1) actions interact only with actions of same level through data;

2) the actions of the levels, except for the lowest one, are a sequence of actions of the immediately lower level, and the actions of the lowest level are a sequence of programming language commands.

See levels of abstraction in details in [9, 10].

5.4.4. Use as high a level of abstraction as possible

There are several aspects to this rule.

One of the abstractions is the formal parameters of methods and constructors. Using formal parameters instead of fields and constants reduces dependencies, since changing fields and constants does not entail changing formal parameters.

Using fields or constants in a method creates the following dependencies:

- to call a method with different values of fields or constants, you must first change these fields or constants, and only then call the method;

- when deleting a field or constant, you will have to change the method.

Another abstraction is abstract classes and interfaces. It is necessary to create dependencies on the most abstract elements (abstract classes, interfaces). The more abstract an element is, the more general entity it describes, and therefore changes less often. If an element changes less often, then all elements dependent on it will also change less often due to changes in this element.

5.4.5. Use dependency injection

Using static objects creates many dependencies on them in different parts of the program, which makes it difficult to modify the program. In this case a static object is like a global variable. Therefore, the use of such objects is undesirable practice.

Dependency injection allows you to eliminate this shortcoming. Dependency injection consists of two stages:

1) creating abstraction;

2) passing the object of this abstraction to a constructor or methods to use instead of static methods.

The abstraction is an interface or an abstract class.
6. Metric for Easily-Modified Programs

To declare that one program is better than another on a certain criterion, a measure must be introduced. There are four main measures of programs:

1. productive: the time to solve the problem and the amount of memory required for this;
2. functional: the number of functions performed by the program;
3. complex: complexity of control and data flows;
4. descriptive (program text parameters): number of program text lines; number of different kinds of statements (control, object-oriented, etc.) in program.

The source lines of code (SLOC) metric is not suitable for comparing easily-modified programs, since this one depends on the programming style and takes into account insignificant parts of the program.

We introduce an NOC (Number Of Commands) metric. NOC is the count of commands in class methods and static blocks. Commands are:

- calls of constructors or methods;
- arithmetic and logical operators;
- headers of the conditional statement (if) and the iteration statements (for, while, do-while);
- the return, break, and continue statements;
- the try statement.

Let’s count the NOC metric for a class (Listing 2 (Listing 4–6 [1])). NOC is counted in the third column, SLOC is counted in the first column.

Listing 2. Class for Counting the NOC Metric
```java
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\W");
                wordCount += words.length;
            }
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } //try
        catch (IOException e) {
            System.err.println("Error:" + e.getMessage());
        } //catch
    } //main
}
```

The NOC metric eliminates the shortcomings of the SLOC metric. However, calculating the NOC metric is more time consuming than calculating the SLOC metric.

The NOC metric can be used in conjunction with the number of methods (NOM) [11]. The smaller the ratio of the NOC / NOM metrics, the shorter the methods and the easier the program is to change. Large NOC / NOM ratio may indicate that method is performing more than one action.
Reducing the ratio of NOC / NOM metrics should not be an end in itself. The metric values are just one of the indicators of an easily-modified program. In medicine, normal body temperature is not an absolute indicator of human health. Diagnostics is carried out in conjunction with other indicators, including quality ones. It's the same with programming. Metrics should be used in conjunction with other indicators, most often subjective, for example, the estimate of the program style by other programmers.

The NOC and NOM metrics are descriptive measures of program.

7. Conclusions
We have made the following conclusions.
I. Existing approaches describe clean code vaguely and emotionally. Therefore, it is necessary to formulate simple and clear criteria and principles of easily-modified programs.
II. The criterion of easily-modified program is «program structure should be such as to anticipate its adaptation and modification» from the Dijkstra’s chapter of the book [6].
III. Easily-Modified program:
- has elements whose naming reflects their purpose in the program;
- is indented for nested operators and blank lines;
- has elements that have one purpose; one assignment matches only one element;
- has the minimum possible number of dependencies.

The criterion and principles are well-known. However they indeed define easily-modified programs.
IV. The number of commands (NOC) metric can be used to compare easily-modified programs.
The ease of modifying the program is a competitive advantage and an indicator of the professionalism of the programmer.

References
[1] Martin R 2009 Clean Code. A Handbook of Agile Software Craftsmanship (New York: Prentice Hall)
[2] Dāsa R 2016 Learn CakePHP With Unit Testing (New York: Apress)
[3] Anaya M 2018 Clean Code in Python (Birmingham – Mumbai: Packt)
[4] Roth S 2017 Clean C++: Sustainable Software Development Patterns and Best Practices with C++ (New York: Apress)
[5] Visser J 2016 Building Maintainable Software: Ten Guidelines for Future-Proof Code (Sebastopol: O’Reilly)
[6] Dahl O-J, Dijkstra E W and Hoare C A R 1972 Structured programming (London: Academic Press)
[7] Avacheva T, Prutzkow A 2020 Workshop on Materials and Engineering in Aeronautics, IOP Conf. Series: Materials Science and Engineering 714 (1) 012001
[8] Gamma E, Helm R, Johnson R and Vlissides J 1995 Design Patterns Elements of Reusable Object-Oriented Software (Reading: Addison-Wesley)
[9] Dijkstra E W 1968 Communications of the ACM 11 341–6
[10] Liskov B H 1972 Proc. of Fall Joint Computer Conf. p 191
[11] Henderson-Sellers B 1996 Object-oriented Metrics: Measures of Complexity (Upper Saddle River: Prentice-Hall)