Applying "On writing well" to Coding

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

William Zinsser has excellent advice and four principles for writing non-fiction. I am amazed how appropriate these principles are to coding. Following these simple principles can make your code better.

On Writing Well

I recently listened to an audio CD from William Zinsser[1]. It is his narration of "On Writing Well." In it he talks about and provides some solid advice on how to write a non-fiction. He presents four simple principles to make an effective writing: clarity, simplicity, brevity and humanity.

While I listened to the tape my mind, as usual, drifted off to think about coding. It occurred to me how relevant these principles are not just to writing but to coding, that one act that we as programmers perform several hours a day.

Writing vs. coding

Mr. Zinsser is specifically talking about writing books, articles and memoirs. It is true that coding is different from writing an article. However, there are similarities as well. In both we are expressing our ideas in written form. In one, we want our readers to understand what we write. In the other, we want our compiler to interpret. However, often we miss the reader who may have to read the code as well. The more I think about it, the more it seems to make sense that these principles are highly relevant. Of course, it is no surprise that some of the good books in software development and refactoring talk about these in one form or another.

Clarity

How often have you scratched your head looking at a piece of code wondering "what on earth is this guy trying to do here." No matter how complicated the logic is you should be able to break it down into understandable code. When you write code, if you are not clear about what you are writing, you certainly can't expect others who read it to understand.

Think about the code you have seen over the years. You would agree that some of the best developers you have come across wrote code that was clear and easier to understand. Reading through their code, you did not have to wonder what they were doing. It flowed naturally.

You probably have also seen your share of code that threatened to make you bald. I once had the opportunity to understand a very critical part of a system by reading through the code. After a few days of effort I had to give up. The code was so convoluted. Some developers have this natural gift to obfuscate their code.
Of course, when you sit down to write the code, and as you evolve it, it will not look pretty. That is the reason you first write the code, make it work and then refactor it to its

better shape and form. If the code is going to be hard to understand, what difference does it make whether it is in source form or binary? You must program with intent and expressively. I call this the ***PIE*** principle.

The next time you see clarify, take a moment to appreciate it. It reminds us that we are looking at code that is understandable, code that is some what different from the ones we are used to.

Next time you write some code, take a look at the code a day or so later. See if you are able to understand it and if it makes easy reading.

Simplicity

A few years ago, programmers did not know enough UML and patterns. Today, they use them too much. A programmer hits a compile button and picks up a magazine to browse. Flips pages and reads about a pattern. Put the magazine down and suddenly the code in front of him appears like the one that can use that pattern he just read about.

Programmers today are compelled to use patterns. If they do not use a few flashy patterns in their code, they feel bad. They think that an object-oriented system must make extensive use of objects.

How about finding a simple solution that works? Each one of us should shed this urge to make things complicated so as to feel better. If you go to a client and present a simple solution, you are worried about loosing the high hopes of the client. A simple solution is something that you can maintain easily. A more complicated one is hard to even understand let alone maintain.

Recently I had a group of people working on designing and developing a game at a symposium. When it came to setting the first player who should play the same, one person suggested

```
        gameBoard.setFirstPlayer(new Player("Venkat"), "X");
```

A player object is first created and it along with the string representing a peg to be placed is sent to the game board object. Another programmer suggested,

```
        gameBoard.setFirstPlayer(new Player("Venkat"), new Peg("X"));
```

While any of the above will work, how about

```
        gameBoard.setFirstPlayerIsX(true);
```

This code is much simpler and does exactly what is needed at this point. It tells the game board that the first peg to be place will be an "X" peg. This is simpler to use, easier to understand as well. A decision as to what to do with a player who wins the game is deferred to later time.

I recently saw a developer who used some complicated patters to solve a problem. He was unhappy with his solution and set out to redesign. After an interactive session of understanding the requirements and walking through the design, he arrived at a solution that was much simpler and used some lighter patterns. He stood up saying this is easy to understand and I can actually code it in the next couple of hours.

I was working on a project when I got drawn into a code that kept on going. The more I wrote, the more it seemed to get complicated. I was quite unhappy about it. I took a quick coffee break, rethought about it sitting away from the computer. That is when I realized that there was a much simpler way to implement it and threw the code out and rewrote it.

Listen to your gut feeling. If you think it is not simple, then it is not simple. There are easier ways to write it and you have to find that code. Patterns are important. However, they have their places and you have to evaluate the options and find the simplest solution that works.

Brevity

This principle can't be overemphasized. Short code that works is better than longer code. Functions must be shorter rather than being longer. Classes must be smaller. Statements must be shorter. What is the point in writing a compound statement that is harder to understand (looses clarity) and is complicated (looses simplicity). Brevity promotes clarity.

If your method is 40 lines of code, consider splitting it into smaller methods by refactoring it[2]. If your class is big, say more than 20 methods, for instance, consider splitting it. Do you have statements like the following?

```
new System.Threading.Thread(new ThreadStart(MyMethod)).Start();
```

or

```
myObj.foo(myObj.f2(3), anotherObj.f3(3.14, 22));
```

Consider splitting it into multiple statements. If you have looked at my code, you will know that I am guilty of this as much as any one. It is so tempting to write code like the above. However, there is certain amount of divineness in clarity and simplicity that comes from brevity.

If you see a code begin repeated over and over (violating DRY principle[3]), refactor the code. If an interface becomes bloated, segregate it (Interface Segregation Principle[4]).

Humanity

I struggled with this for a while before I realized how I missed the point of its relevance to coding. It is one of the major problems in coding. The reason so much code is hard to understand is that you are writing it for the compiler, the runtime, the processor. You should think about the human (one being yourself some time in the future) who has to read it. Often times, we think of program as this abstract, inhumane thing that is executed by this lifeless machine.

In a recent interactive development activity I wanted to know the next step I should program. First every one in the group threw a blank. Then a few members started throwing in suggestions that was all over the map. At this point I pulled a person out of the group and asked, “OK, you and I are going to work with this system. What do you want to do next?” He came with the right answer instantaneously. Putting him and me in there gave the perspective and clarity to the issues. It was a difference between day and night.

If you write the code with the mind set that the machine will execute it, the code no wonder turns out dry, boring and so hard to understand. On the other hand, if you think of this as a person, a real super fast human executing, then there is life in your code. The code tends to look more understandable and meaningful.

I often time think of myself as an object and think of methods being executed on me. You realize that abstraction and modeling requires quite a bit of imagination. This imagination becomes easier and produces better results when you put yourself in there instead of a machine.

Principles on coding well

Mr. Zinsser gave us more than just principles for writing. His principles are very relevant and applicable for coding as well. Just following his four simple principles - clarity, simplicity, brevity and humanity can improve your code. Think about these the next time you punch those keys.

Your feedback

Tell it like it is. Did you like the article; was it useful, do you want to see more such articles? Let us know, as that will motivate us to continue writing. Did you not like it? Please tell us so we can improve. Your constructive criticism makes a difference. Do you have suggestions? Please send those to use at agility@agiledeveloper.com. Thanks for being a subscriber to the free Agility news letter.

References

1. “On writing well,” audio CD by William Zinsser (look at the book by the same name as well).
2. “Refactoring Improving The Design Of Existing Code,” Martin Fowler, Addison-Wesley.
3. “Pragmatic Programmer,” Andy Hunt and Dave Thomas.
4. “Agile Software Development, Principles, Practices and Patterns,” Robert C. Martin, Prentice Hall.