Prof. Mann's Rules of Programming

Golden Rule:

It must work

Meaning: The program must do what it was intended to do, regardless of the algorithm or approach you selected to solve the problem. Testing helps to validate this is true. Simply executing is insufficient; just because it runs doesn't mean it's correct.

Silver Rule:

Give the user a good experience (or, don't frustrate the user)

Meaning: make it easy for the user to provide your information (example: put prompt into field). Minimize the number of mouse clicks. Don't make the user repeat multiple entries when only one entry is invalid. Give useful information in any output. Not "invalid input" but "Invalid input of xxx received, expecting yyyy"

Bronze Rule:

You will read more code than you ever write so write readable code

Meaning: Use good variable names. Insert appropriate, but not excessive, comments that add to understanding of your code. White space is free so use it generously, it makes a huge difference in reading code.

Copper Rule:

Expect the user to be an idiot (or ID10T if we're trying to be PC)

Meaning: Program defensively. Coding the "happy path" is relatively easy; it's everything else that poses the greater challenge.

Pewter Rule:

Effective communication is essential

Meaning: Ensure the listening party has heard and understood all you've stated, don't assume!

General guideline: under promise and over deliver.

Practice iterative development. Make it work, then make it better

From a friend of mine who is a Software Engineer. I asked him to review my rules and if he had any additional input. The remainder of this is his response.

I would agree that the 4th rule might be more important than the third. Defensive programming is very important in the real world, not only to validate user input but also to defend against method callers. It also is part of the first and second rules since it must work even with unexpected data. The best example I can think of is null reference issues - you make a call to a method that can return an object or null - in that case you must check for null before referencing the object or an exception is thrown.

So, the rules 1, 2, and 4 are operational rules - they apply to the program's operation. Rule 3 is more lexical, which is important but does not directly impact the program operation. But clean, well-written code generally results in a better and easier to maintain program. Thus rule 3 might be a separate category.

There are a couple of things that go along with readable code.

```
if(int i=0;i<10;i++)   // bleh mashed together and hard to read
  if (int i = 0; i < 10; i++)  // much easier to read
```

Use descriptive names for variables but do not get too crazy with it.

- hasPreAuthorization (good)
- hasAuth (not so good)
- hpa (bad)
- thisTransactIonHasAPreAuthorizationThatIsAvailable (evil)

loop iterators or very small scope transitional variables are ok to be abbreviated.

```
for (int i = 0; i < 10 i++)
```

idx for index etc.

Make methods small and concise ... long methods are difficult to read and harder to understand.

Make classes specific and named clearly.

Comments - document the -why, not what. The code is there to be read and understood but the intention is not.

Magic numbers ... document constants using a CONST, define, or enum etc. I have actually seen code that looks like

```
var someVar = document.Height + document.Width * 2(1.4 + 5 - 14);
```

yes, what do these numbers MEAN??

And finally - SPELLING COUNTS! Spell things correctly. Yes, it matters, and yes, engineers can spell correctly (I have heard the negative to both those points, grrr). I have actually refactored a number of classes after the third time typing a misspelled variable.

There are a couple books that I can suggest for students

Code Reading by Diomidis Spinellis (excellent examples on how to read code).

Enough rope to shoot yourself in the foot by Allen I Holub. A practical set of "rules" for C / C++ programming.

Clean Code (ms press) ... there are a couple in this series.

1. Always write your code and comments as if the maintenance engineer is a sociopath with your home address, phone number, all your email addresses, and bank account information

2. Do not prematurely try to optimize. Make it work, then make it perform.

3. Do not add features that you think you might need in the future. Make your code maintainable and extensible but only add what you need now, not what you think you might need in the future. If it's extensible it can easily be added.

4. Do not be clever. Clever code is rarely clean, maintainable code.