Tutorial for clean code

Disclaimer:

This tutorial is about refactoring bad code into clean code. Clean code is subjective and you may will change some parts in the final code to get better clean code in your opinion. The code presented here isn't perfect. However, you should include these tips and tricks in your coding routine, so you won't have to refactor this bad code in the end of your project. It is much easier to refactor when the code isn't that bad. This leads to another helpful point: always "tidy up after yourself". Once you start to leave one or two things unfixed, it becomes much easier to leave „just one more". You want to learn how to write clean code? Then keep these points in mind whenever writing and refactoring your own or someone else's code.

Starting point:

The program is a simple game called "Galgenmännchen". Therefor, the game host types in one word or (when you want to make it more complicated) a whole sentence. Each round the other players are asked to type in a letter (to simplify the program there is no exception when the user types in more than one letter, just the first letter will be used). After every input, the users get the current result: What they guessed correct so far and how many mistrials there are left. There are two possible endings of the game: The host can win the game when there are no mistrials left or the users can win when they guessed the correct word.

```
Welcome to the game 'Galgenmaennchen'!
Please let the game host enter a word first:
sunflower
Please enter each round a letter of your choice. Try to guess the word!
(If you type in more than a letter then the first letter will be used.)

Letter          Output          Left mistrials
a
                ---------               9
e
                -------e-               9
o
                -----o-e-               9
l
                ----lo-e-               9
```

The whole project consists only of one class and only of one method, the main method of the class Galgenmaennchen. There are several comments in the code, that the developer understands what is going on. Please take a deeper look at this commit https://github.com/adamcel/CleanCode/commit/f6ba261d9f442d9a15ec2f20c5b24cc5831dfb7d which was the initial commit. Can you understand the code right away? Or do you have to think about it? That's the point when you know that you have to refactor.

Step 1: Writing a solid test suite

When writing clean code, it is important to think about the code you are writing. The best way to do so, is writing tests because you have to take some time and rethink your code. This may sound to be more time consuming when you are writing just a simple function and have to include a test but as you go along you are getting faster in writing tests. It also will save you a lot more time when finding bugs and fix them.

Before you start to refactor, you have to make sure that you have a solid test suite. This will prevent you for breaking any functionality of your code. Since the example we're working with only consists of one method right now and we would have to write one big test for the whole program, we will skip this step. But keep in mind, that this is always the first step when refactoring.

The following steps can be done in a different order. All steps can also be repeated as often as you think you need them.

Step 2: Extract methods

Extract methods means that you should break large methods into smaller pieces. Clean code often comes with the definition of methods that they have to do only one thing. This single-responsibility principle also will help to comment your code without even writing a real comment. For commenting you can use the method and variable names. Our program is the best example for this step. Since there is only one method, we can break it totally into smaller pieces.

https://github.com/adamcel/CleanCode/commit/4ee5aa84675bcfea2b26ec4add34533cfe714110

View this commit to get a better inside in what happened here. The first step was to move the total functionality of the game into a new method. This will help in a later commit to transfer the main method into another class. The next step was to remove the break statements. Among others, four methods were extracted: start, explainGame, garbleWord and checkIfNewLetter.

Step 3: Start the game from another class

In the next commit the main method was transferred into a new class, called Main. Therefor, the method startGame was changed from private to public.

https://github.com/adamcel/CleanCode/commit/46eae2f70cfddd591bc0e25e19fc733ef417fe6c

Step 4: Removing temporary variables

This step was about removing temporary variables and use them as global variables instead. This also helps us to reduce dependencies on other methods because less methods need parameters passed.

https://github.com/adamcel/CleanCode/commit/7c9752520f0296bda62a095443c16cf48a5e5e75

With this step you might be a little bit careful. It isn't the best way to use only global variables. Sometimes even local variables and passing parameters can be even better. Just think about every variable before you decide to use a local/temporary or a global one. Your IDE or Codacy might even prefer temporary variables instead of global ones if you only use them in one method.

Step 5: Renaming methods and variables

This step is very important for you or the next developer when you have to understand the code again and maybe also change something about it. In the example, I changed the variable word into solution because it represents the solution of the riddle. Moreover, I changed the name of the method start() accordingly into readSolution.

https://github.com/adamcel/CleanCode/commit/03e75201e37b7181e3eb0e13b282016703490492

Step 6: Repeat the previous steps

This step is self-explaining. To achieve clean code, you can't do every step at a time in just one commit. You have to go back and repeat the steps.

First, I removed temporary variables again and used the already existing global variables instead. Therfor, the methods readSolution and garbleWord saved their temporary values in global variables. This made it possible that those methods changed from returning Strings into void.

https://github.com/adamcel/CleanCode/commit/2602ad5d5d8fc152bb1c9fcafd01808241ef3d14

Next step was to extract methods again. Those methods are now called updateGuessedWord, calculateLeftMistrials and checkForCorrectAnswer. This step also reduced some comments because the method names are used as implicit comments.

https://github.com/adamcel/CleanCode/commit/59daf6da354fc297c932cecf8be4a38f221db8e9

Moreover, step 5 was repeated. Therefor, the global variable guessedWord was renamed into answer, as well as all other local variables and methods accordingly. This lead to a better understanding as well as to shorter variables and method names.

https://github.com/adamcel/CleanCode/commit/1c9d3cd4004558269d66aa45ebe9e47e3e94b53b

This is the last commit on repeating any of the previous steps. This commit was again about extracting methods. The extracted methods of this commit were printGameStatus and endGame. Both print something onto the console: printGameStatus prints the current game status like the current guessed answer and the left mistrials. endGame checks if there are any mistrials left. If so then it prints that the host wins, if not then it calls the method checkForCorrectAnswer.

https://github.com/adamcel/CleanCode/commit/04fc838e006c2ed28a45ff2abf6ebf7d1790e33f

Step 7: Switched to using StringBuilder

This commit was to remove appended Strings, especially in loops. Strings are generated only one time. When you change a String, your computer generates a new String and set the pointer to the new address. Changing a String in a loop (often appending) leads to generating many Strings which are never used later on. By using a StringBuilder, your computer generates only one String and reserves more memory. You can then append the String and use this extra memory. Can you use StringBuilders in your own project?

https://github.com/adamcel/CleanCode/commit/7b51d10d32d6d2cfde5c76ea4368020fb5b38f27

Further steps:

Since this was only a small example, there are still a few refactoring tips and tricks that can help you in your own project. Those are:

- Move methods to their corresponding classes
- Reducing complexity of classes and methods (e.g. with polymorphism)
- Shorten parameter list
- Transform error codes into exceptions

If you want to get a better inside in refactoring, then take a look at: Refactoring by Martin Fowler

The steps presented in this tutorial are further explained into this book. Chapter 1 gives also another example on refactoring.