Seminar

Effiziente Programmierung

Code Layout, Code Style and Documentation

Christopher Jarsembinski

Betreuer:
Konstantinos Chasapis

March 15, 2017

1 Prolog

In the following pages I will try to elaborate the importance, variety and problems that come with layout, style and documentation of code. It will be a brief overview and no in depth analysis, but should give a general understanding of the topic and why it is worth thinking about this, before starting larger projects, especially in groups or teams.

Abstract

Programming itself is often a difficult task, thinking about the existing problems and trying to abstract them so that you can solve it with a computer. Many forget though, that before you **start to work on** that solution, you should think about **how to work on** that solution. A programm solves its task, but a great programm is expandable, adaptable and usable by others than the author itself. To achieve this I will explain some methods, ways and problems you should consider for yourself and others who help you now or may work on your project in the future. These involve appearance,readability, understandability of your code and the possibility for others to reconstruct your thoughts or atleast get why you use certain ways to solve problems.
It is important to take these points into consideration before you start programming, but you should also remember them while you are working on the project and even after finishing your project. None wants to read thousands of lines of code before even knowing if that would help him with his problem or if he can include that in his own work. So I will show some tools and ideas that help you achieve somewhat of a sneak peek or overview of your project so that others can easily understand its use or see its advantages in comparison with other approaches to this topic or problem.

2 Code Layout

2.1 Purpose

Code layout is the general appearance of your code. It is an important aspect of writing code for humans and that is what you have to keep in mind, you don't only write for the computer but for humans as well. For the compiler it doesn't really matter whether or not you use indents, whitespaces, white- or multiple lines, but writing code is not only functionality. These ideas might be obvious or easy, but are nontheless important to assure that your code is usable by humans, because it is easier to overview your own code for yourself and others. In a matter that structures and connections are easily seen.

2.2 Utilization

Now that we know why it should be done, let's see how to do it. This is an example of properly written code:

```
class HelloWorld
    {
    public static void main(String[] args)
        {
        System.out.println("Hello World");
        }
    }
```

So a little step back from what we see to what it could be without any thought put into this. Let's begin with a programm that works and is no problem for compilers, but humans might have a problem understanding it.

```
class HelloWorld{public static void main(String[] args){
System.out.println("Hello World");}}
```

As mentioned, this code is totally fine for the compiler, but a programmer not familiar with this code, would need some time figuring out what is written there, even though this example is not complex in any way. So now let's take the first step into fixing this mess.
One command should be one line, there might be some languages that support ways to combine commands into one line, but the readability will suffer.

```
class HelloWorld{
public static void main(String[] args){
System.out.println("Hello World");}}
```

Even though the change was small, it is far easier to understand our programm. On first glance we can see the command sequence and step by step what the programm does in a timely manner. Problems that would appear now would be loops.

```
for(int i=1; i<11; i++){
System.out.println("Count is: " + i);
System.out.println("Next Count is: " + (i+i)}
```

We can see the sequence, but we can't see immediatly where the loop starts and ends, we have to search for the brackets. So the next step is spending one line for brackets.

```
class HelloWorld
{
public static void main(String[] args)
{
System.out.println("Hello World");
}
}

for(int i=1; i<11; i++)
{
System.out.println("Count is: " + i);
System.out.println("Next Count is: " + (i+i)
}
```

The class, method structure is now easy to get, the loop is clearly visible and easier to put into context with further commands. Now we can add indents to see which which methods belong to which classes and to avoid these stacked brackets at the bottom, because it is pretty difficult to sort these to brackets in the whole script. So lets put the loop and the earlier programm together.

```
class HelloWorld
    {
    public static void main(String[] args)
        {
        System.out.println("Hello World");
        for(int i=1; i<11; i++)
            {
            System.out.println("Count is: " + i);
            System.out.println("Next Count is: " + (i+i)
            }
        }
    }
```

Even though the programm is now more complex than in the beginning it is easy to grasp and to understand, just by some easy steps. This is solely about the appearance so far, kind of the beauty aspect. Important to mention is that it is partly dependent on the editor you use, because indentation varies from 2 to 4 to 8 whitespaces, some prefer a darker background and the font style and size can vary too.

3 Code Style

3.1 Purpose

The style is less about the general appearance and more about a logical structure and patterns in your code for others to find and follow. There is no optimal style, but it is important to stay in one style, since you normally don't work alone on your programm or others want to expand it and once they understood the style you use, they can adapt to it and keep it for segments they will implement. This will create an overall coding style that makes it easier to understand if others read it, since they don't have to get accustomed to the habits of multiple programmers but rather to one style these programmers agreed upon. So again the coding style is solely important for humans and not necessary for the compiler.

3.2 Glimpse of Styles

As I mentioned, there is no optimal style, just the one, that fits your way the best. So I won't tell you to take this or that style, but rather show you some and let you see for yourself what you like. Keep in mind that you can find many more styles if you just look for them.

3.2.1 Indentation

Indentation is mainly about the position of brackets around your commands. Should they be aligned to it? Should they have their own lines? These could be questions you can ask yourself to determine that.

The pattern I will use for the examples will be as follows:

Stylename

```
Some Code
{
}
```

K&R

```
while (x == y) {
    something();
    somethingElse();
}
```

Ratliff

```
while (x == y) {
    something();
    somethingElse();
    }
```

Allman

```
while (x == y)
{
    something();
    somethingElse();
}
```

Whitesmiths

```
while (x == y) {
    {
    something();
    somethingElse();
    }
```

GNU

```
while (x == y)
  {
    something();
    somethingElse();
  }
```

Horstmann

```
while (x == y) {
{     something();
      somethingElse();
}
```

3.2.2 Spaces

The spaces between variables and operators or symbols is something worth thinking about as well. Not using spaces could work, if your editor properly highlights the different characters or if the values are short.

```
int i;
for(i=0;i<10;++i)
    {
    System.out.println("%d",i+i);
    }
```

Looking at this example it is readable, but also obvious that you need some time to find the seperating semicolons. Now let's see what happens, if we split up the single expressions with spaces.

```
int i;
for(i=0; i<10; ++i)
    {
    System.out.println("%d", i+i);
    }
```

It is easier to see the single expressions and considering that these expressions are not too complicated themselves. Eventually we will now split the single values and operators.

```
int i;
for(i = 0; i < 10; ++i)
    {
    System.out.println("%d", i + i);
    }
```

Each value, variable and operator is clearly visible, it may be overdone for this example, but once the bound values get more complicated, considering these spaces might be a good idea.

3.2.3 Alignment

While initializing your variables or lists, considering to align the names or different elements can help to improve the overview and readability in some cases.
This choice may differ over several projects you start, depending on complexity and variety of variable types and list length.
Lets start with variables:

```
\\No Alignment
int i;
long longer;
String sentence;

\\Simple Alignment
int    i;
long   longer;
String sentence;

\\Alignment + Tabs
int         i;
long        longer;
String      sentence;
```

If you don't initialize too many variables, no alignment should be fine, but not only the number of variables increases the time you need to comprehend them does too. Simple alignment will almost always work, but can be tricky if you add longer types, since you have to align the earlier names accordingly. Now we get to the benefit of alignment with tabs. Even if you add longer types, the tabs you precautiously placed should prevent any mess up, but it could happen that the gap between an integer type and the name gets too big, to properly connect them by looking at it.

```
\\Alignment + too many Tabs
int                                         i;
thisTypeIsTooLongButHelpsThisPurpose        someName;
```

Thinking about these styles or adapting to one can be difficult, especially if you work on multiple projects with different styles. Luckily there exist tools, that support you in your coding choice and even automatically format your code or remember these for you.
The web offers you multiple formatters that are free to use and there exists and even wider variety of editors that support you in this cause, e.g. Atom or Eclipse. Most editors provide support for multiple languages, but you can also find specialized ones.

4 Code Documentation

4.1 Purpose

Once you finished your programm and everything is checked and works as intended, the next step occurs, bringing this programm to the users or fellow programmers who want to use or adapt what you created.
Now change your perspective and be one of these people that look for your programm and manage to find it, they encounter thousands, maybe tenthousands of lines of code and have to see if this is what they look for. Would you look into all of this without knowing whether it is worth it?
That's why code documentation is important, it is a way for yourself or others to easily understand huge programms or bigger segments of code without having to read all of it and understand all the ideas used. Code documentation includes writing comments, FAQs, a readme.txt and even bigger overviews without showing specific implementations.

4.2 Utilization

4.2.1 Comments

So as mentioned, the documentation is some kind of guide to your programm. To show what I mean with that, the following example is code I wrote, but without comments. The purpose of this segment is to sort a list in a specific order, considering which game is played to display it in a box on the screen.

```
def sortedList(self):
    self.sortList=sorted(self.sortList,key=lambda x: int(x[1]))
    if self.Ts.gameMode == "TDE":
        self.sortList = self.sortList[::-1]
```

Without any comments all of this seems pretty strange. What we can grasp so far is that we sort a list and if we look into this a bit more, we can see in line 2, that we sort by the second element of each elementpair, so we should have a list of lists? Now after that happened we check for some gamemode and reverse the list if needed to. Let's try this with comments:

```
##Sort given list after second tuple value and display it in the listbox
def sortedList(self):
    ##D&D Initiative Order
    self.sortList=sorted(self.sortList,key=lambda x: int(x[1]))
    ##Change intiative order, considering the gameMode
    if self.Ts.gameMode == "TDE":
        self.sortList = self.sortList[::-1]
```

The comments make this far easier to understand and to put in context with other segments. We don't even have to invest too much thought into it, since the comments explain why this is written. ##Important side note here, comments should always explain **why** you do something, not **how**. If I want to know how you did it, i can read the code.

4.2.2 Readme

Writing a readme is necessary, so you can tell people about important facts or steps of your programm, e.g. installation, usage, FAQs or the license under which this is allowed to use. You prevent people to get confused and stop them from turning away, because they are unsure. So what should a readme contain?
First it should be a sneak peek of your programm. You can shortly describe what your programm does and what makes it different or better than others. You could also explain in some sentences why you chose to write this instead of taking an existing one.
Answer some frequently asked questions, short FAQs. If many people ask the same thing, it seems to be important, so prevent them from waiting by including these and keep them up to date.
Instructions on how to use or install the programm or how people can contribute or contact you is always a good idea.
And eventually the license under which this can be used. You should use one of the common ones, so it does not need too much explanation. Some examples would be: GNU General Public License, BSD License, Public Domain

And again, there do exist tools, that assist you in documenting your code or in creating helpful options to make your code or programm more approachable. An example would be Doxygen, that can automatically analyze your code and summarize it in numerous ways, given that you wrote your code properly. With written probably it is meant that you follow certain patterns, so Doxygen understands what you want it to do. For example in python comments are marked with "#", Doxygen only uses comments that are written as "##". This enables you to comment as usual, to leave TODOs or comments for yourself or members of your team, without worrying of forgetting them when you use Doxygen, since Doxygen simply ignores them. These "marks" differ for each language, but Doxygen is well documented, so it is easy to find the proper syntax. Here is an example of how documented code would be displayed by Doxygen:

Examplecode:

```
##Mainwindow Functions // ignoreToken is to ignore the eventtoken
    def addCharacter(self,ignoreToken):
        #Pattern
        pattern=re.compile("[A-Za-z]+[0-9]*[,][0-9]+\Z")

        if(pattern.match(self.CName.get())):
            #Refine given String
            refinedString = self.CName.get().split(",")
            for x in range(1,len(refinedString)):
                refinedString[x]=refinedString[x].rstrip()
                        ...

##Handle HP
    def healHP(self,ignoreToken):
        if not self.healCheck:
            self.healCheck = True
            healWindow = HPChangeWindow(self)
            self.HPcolor()

    def dmgHP(self,ignoreToken):
        if not self.dmgCheck:
            self.dmgCheck = True
            dmgWindow = DmgChangeWindow(self)
            self.HPcolor()

    ##determine HP color, red < 0, yellow=0,green > 0
    def HPcolor(self):
        name = self.getName()
        if (self.HPDic[name] < 0):
            self.HP.config(fg="red")
        elif (self.HPDic[name] > 0):
            self.HP.config(fg="green")
        else:
            self.HP.config(fg="yellow")
```

Doxygen:

Public Member Functions

def	**__init__** (self, selector)
def	**addCharacter** (self, ignoreToken) Mainwindow Functions // ignoreToken is to ignore the eventtoken.
def	**healHP** (self, ignoreToken) Handle HP.
def	**dmgHP** (self, ignoreToken)
def	**HPcolor** (self) determine HP color, red < 0, yellow=0,green > 0

Clearly visible is that normal comments are ignored, the amount of code displayed is reduced to what is needed to get a gist of the purpose and all this just by running a programm.

5 References

- http://tech.dolhub.com/article/computer/code-Layout
- https://www.stat.auckland.ac.nz/~paul/ItDT/HTML/node20.html
- http://www.liquisearch.com/indent_style/ratliff_style
- http://www.liquisearch.com/indent_style/kampr_style
- http://www.liquisearch.com/indent_style/allman_style
- http://www.liquisearch.com/indent_style/whitesmiths_style
- http://www.liquisearch.com/horstmann_style
- http://www.liquisearch.com/gnu_style
- http://www.writethedocs.org/guide/writing/beginners-guide-to-docs/
- http://codebeautify.org/python-formatter-beautifier
- https://www.stack.nl/~dimitri/doxygen/manual/starting.html
- https://atom.io/