

04834580 Software Engineering (Honor Track) 2025-26

# Code Style & Documentation

Sergey Mechtaev

[mechtaev@pku.edu.cn](mailto:mechtaev@pku.edu.cn)

School of Computer Science, Peking University



*There are only two hard things in Computer Science: cache invalidation and naming things. — Phil Karlton*



According to Benjamin Lee Whorf, **thoughts** are determined by the specific grammar and **vocabulary** of the language in which ideas are expressed.



According to Edsger W. Dijkstra, “the separation of concerns”, even if not perfectly possible, is yet the only available technique for effective ordering of one’s **thoughts**.

*The function (including its name) can capture our mental chunking, or abstraction, of the problem. [1]*

```
def process_string_explicit(s):
    while s and s[0] == ' ':
        s = s[1:]
    while s and s[-1] == ' ':
        s = s[:-1]

    if len(s) > 2:
        start = 1
        end = len(s) - 1
        middle = s[start:end]
        middle_upcased = middle.upper()
        s = s[0] + middle_upcased + s[-1]

    return s
```

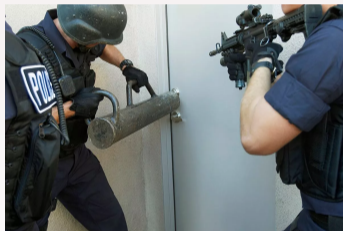
```
def process_string_trim(s):
    s = s.strip()

    if len(s) > 2:
        start = 1
        end = len(s) - 1
        middle = s[start:end]
        middle_upcased = middle.upper()
        s = s[0] + middle_upcased + s[-1]

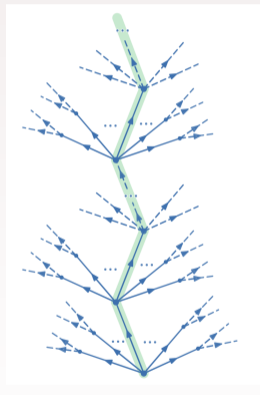
    return s
```

The boundaries of procedural abstraction are determined by the meaning of “strip”.

A general problem-solving technique that consists of systematically checking all possible candidates for whether or not each candidate satisfies the problem's statement.



**Tree** is hierarchically organized data structure, where from the **root** item the other items **branch** out into nodes and **leaves**. A collection of trees is often called a **forest**.



## Definition

Linguistic Antipatterns (LAs) in software systems are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding.

## Definition

Creating conflicting methods or functions regarding their functionality and naming.

```
def getFoos() -> Foo:  
    ...  
    foo: Foo = ...  
    return foo
```

```
def isGoo() -> str:  
    ...  
    return 'yes'
```

```
def setValue(self, value) -> Any:  
    ...  
    return new_value
```

```
def getFoos() -> list[Foo]:  
    ...  
    foos: list[Foo] = ...  
    return foos
```

```
def isGoo() -> bool:  
    ...  
    return True
```

```
def setValue(self, value) -> None:  
    ...
```

## Definition

Method or function names that have binary operators like AND and OR are possible violators of the single responsibility principle.

```
def render_and_save():  
    # render logic  
    ...  
    # save logic  
    ...
```

*// "a" could mean anything*

```
const a = 5
```

*// "Paginatable" is a broken English*

```
const isPaginatable = a > 10
```

*// Hard to read*

```
const onItmClk = () => {}
```

*// Does not reflect expected result*

```
const isEnabled = itemCount > 3  
return <Button disabled={!isEnabled} />
```

*// Units are unclear*

```
int fileSize
```

*// Redundant meta-data*

```
String valueString
```

*// Better*

```
const postCount = 5
```

*// Better*

```
const hasPagination = postCount > 10
```

*// Better*

```
const onItemClick = () => {}
```

*// Better*

```
const isDisabled = itemCount <= 3  
return <Button disabled={isDisabled} />
```

*// Better*

```
int fileSizeGb
```

*// Better*

```
String value
```

## Definition

Is a set of rules or guidelines used when writing the source code for a computer program.

- ▶ Reduce the number of choices a developer has to make.
- ▶ Makes code more readable and predictable.


```
if(x < y) {  
    x= y;  
    y= 0;  
} else {  
    x= 0;  
    y= y/2;  
}
```

```
if (x < y) {  
    x = y;  
    y = 0;  
} else {  
    x = 0;  
    y = y / 2;  
}
```

Poorly formatted code distracts attention, takes longer to read.


```
// Eliminate whitespace from the beginning of t.
while (t.length() != 0 && isWhitespace(t.charAt(0))) {
    t= t.substring(1);
}

```



```
// If t is empty, print an error message and return.
if (t.length() == 0) {
    ...
    return false;
}

```



```
if (containsCapitals(t)) {
    ...
}

```

Use blank lines to separate logical parts of your algorithm.

```
if (condition) {  
    ...  
    30 lines of code  
    ...  
} else single-statement
```

```
if (!condition) {  
    single-statement  
} else {  
    ...  
    30 lines of code  
    ...  
}
```

Put the shorter of the then-part and else-part first.

## Definition

Written text or illustration that accompanies computer software or is embedded in the source code.

Two main categories:

- ▶ System documentation.
- ▶ User documentation.

Not descriptive:

```
List<Triple<Integer,Integer,Integer>> p = f();
```

More descriptive:

```
List<Point3D> path = findShortestPath();
```

Using good names, types and abstractions make code self-documenting.

```
@Test
public void testWrapNull() {
    assertEquals("", wrapper.wrap(null, 10));
}

@Test
public void testOverTheLimitShouldWrapAtSecondWord() {
    assertEquals("word word\nword",
        wrapper.wrap("word word word", 5));
}

@Test
public void testLongerThanLimitShouldNotWrap() {
    assertEquals("word word", wrapper.wrap("word word", 6));
}
```

Useless comments (antipattern):

```
/* set age to 32 */  
int age = 32;
```

Useful comments:

```
function addSetEntry(set, value) {  
    /* Don't return `set.add` because it's not chainable in IE 11. */  
    set.add(value);  
    return set;  
}
```

```
class InputStreamReader {  
    int read(char[] cbuf, int offset, int len) throws IOException  
    ...  
}
```

Important aspects of method documentation:

- ▶ How to call the method.
- ▶ What the results are.
- ▶ What the effects are.

```
class InputStreamReader {  
    int read(char[] cbuf, int offset, int len) throws IOException  
    ...  
}
```

How to call the method:

**Arguments** cbuf is non-null, offset is non-negative, len is non-negative, offset + len is at most cbuf.length

**Input state** the receiver is open

```
class InputStreamReader {  
    int read(char[] cbuf, int offset, int len) throws IOException  
    ...  
}
```

What the results are:

- ▶ The method returns -1 if the end of the stream has been reached before any characters are read.
- ▶ Otherwise, the result is between 0 and `len`, and indicates how many characters have been read from the stream.

```
/**  
 * This method reads up to length characters from the stream into  
 * the specified array starting at index offset into the  
 * array.  
 *  
 * @param buf The character array to receive the data read  
 * @param offset The offset into the array to start storing characters  
 * @param length The requested number of characters to read.  
 *  
 * @return The actual number of characters read, or -1 if end of stream.  
 *  
 * @exception IOException If an error occurs  
 */  
public int read(char[] buf, int offset, int length) throws IOException
```

```
public int read(char[] cbuf,  
               int offset,  
               int length)  
    throws IOException
```

Reads characters into a portion of an array.

**Specified by:**

read in class Reader

**Parameters:**

cbuf - Destination buffer

offset - Offset at which to start storing characters

length - Maximum number of characters to read

**Returns:**

The number of characters read, or -1 if the end of the stream has been reached

**Throws:**

IOException - If an I/O error occurs

**Description:** A clear and concise textual explanation of the class, method, or field. Intended to help developers understand its purpose and usage.

**Block Tags:** Metadata annotations that provide specific details about the class or method. Common tags include:

- ▶ `@author`: Specifies the author(s) of the code.
- ▶ `@version`: Describes the version of the code.
- ▶ `@param`: Documents method parameters.
- ▶ `@return`: Explains the return value of a method.
- ▶ `@exception` (`@throws`): Details the exceptions thrown by a method.
- ▶ `@see`: Provides references to related resources (classes, methods, etc.).
- ▶ `@since`: Indicates when a specific feature was introduced.
- ▶ `@serial`: Used to document serializable fields or methods.
- ▶ `@deprecated`: Marks elements that are outdated and suggest alternatives.

## Useless JavaDoc (antipattern):

```
/**  
 * Sets the tool tip text.  
 *  
 * @param text the text of the tool tip  
 */  
public void setToolTipText(String text) {
```

## Useful JavaDoc:

```
/**  
 * Registers the text to display in a tool tip. The text  
 * displays when the cursor lingers over the component.  
 *  
 * @param text the string to display. If the text is null,  
 * the tool tip is turned off for this component  
 */  
public void setToolTipText(String text) {
```

## Definition (Ambiguity)

The quality of being open to more than one interpretation; inexactness.

```
def triples_sum_to_zero(l: list):  
    """Takes a list of integers as an input. Returns True if there are three  
    distinct elements in the list that sum to zero, and False otherwise."""  
    ...
```

- [1] Allen Downey et al.  
How to think like a computer scientist: learning with python.  
2008.
- [2] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol.  
Linguistic antipatterns: What they are and how developers perceive them.  
*Empirical Software Engineering*, 21:104–158, 2016.