R FOR HUMANS

PETER HURFORD

FEBRUARY 15, 2018

WHY BE READABLE?

Should a developer aim for readability or performance first? [closed]

48

11

Oftentimes a developer will be faced with a choice between two possible ways to solve a problem -- one that is idiomatic and readable, and another that is less intuitive, but may perform better. For example, in C-based languages, there are two ways to multiply a number by 2:

```
int SimpleMultiplyBy2(int x)
{
    return x * 2;
}
```

and

```
int FastMultiplyBy2(int x)
{
    return x << 1;
}
```

The first version is simpler to pick up for both technical and non-technical readers, but the second one may perform better, since bit shifting is a simpler operation than multiplication. (For now, let's assume that the compiler's optimizer would not detect this and optimize it, though that is also a consideration).

As a developer, which would be better as an initial attempt?

Readability 100%

14

If your compiler can't do the "x*2" => "x <<1" optimization for you -- get a new compiler!

Also remember that 99.9% of your program's time is spent waiting for user input, waiting for database queries and waiting for network responses. Unless you are doing the multiple 20 bajillion times, it's not going to be noticeable.

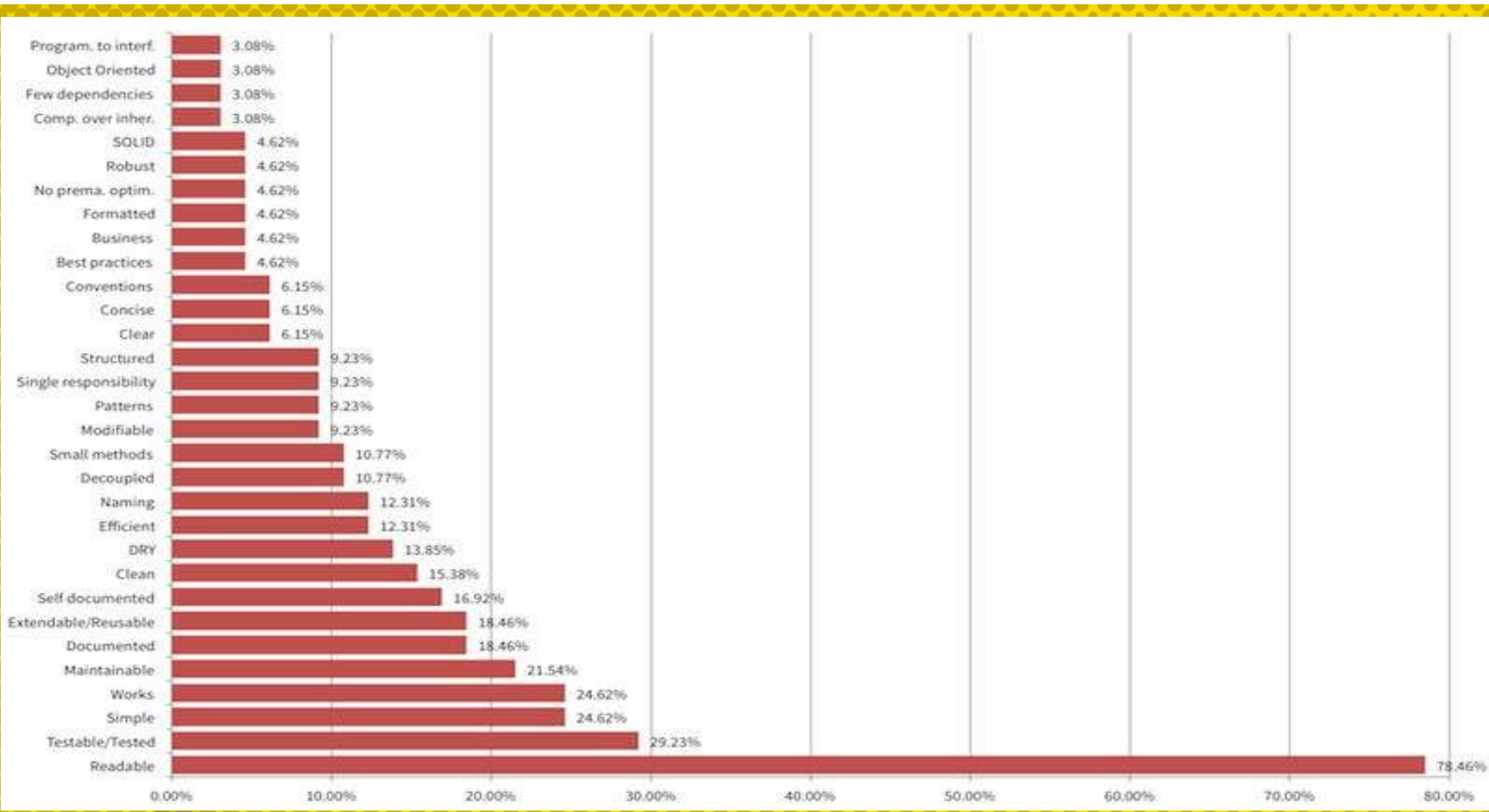
4

A often overlooked factor in this debate is the extra time it takes for a programmer to navigate, understand and modify less readible code. Considering a programmer's time goes for a hundred dollars an hour or more, this is a very real cost.
Any performance gain is countered by this direct extra cost in development.

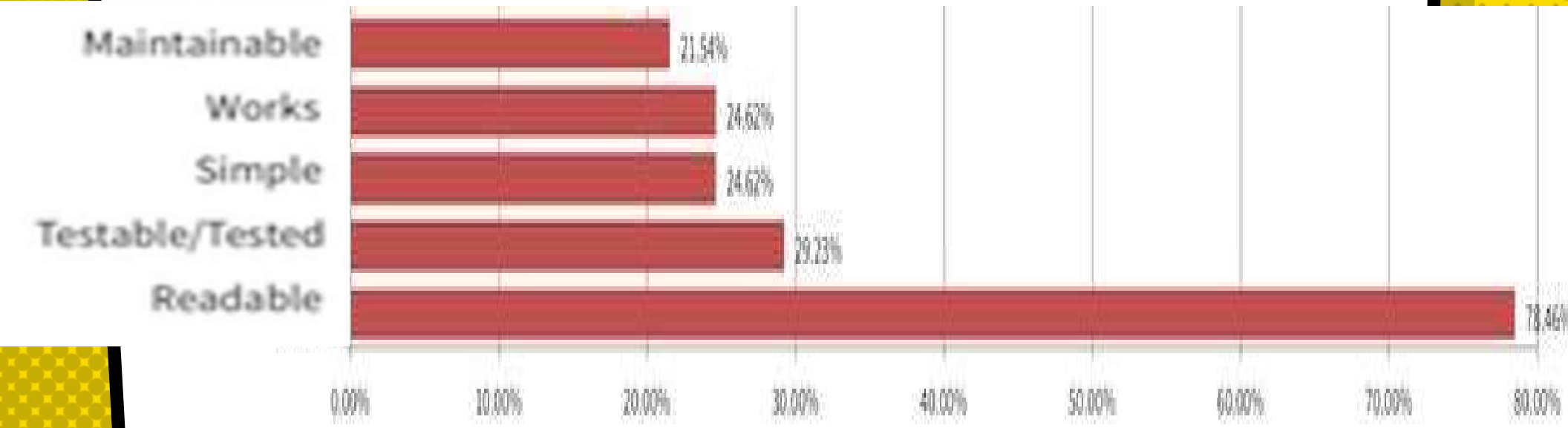
What is good code? A scientific definition

- 65 developers, sampled by convenience
- Asked the same open-ended question: **"What do you feel makes code good? How would you define good code?"**
- Answers grouped into 31 groups

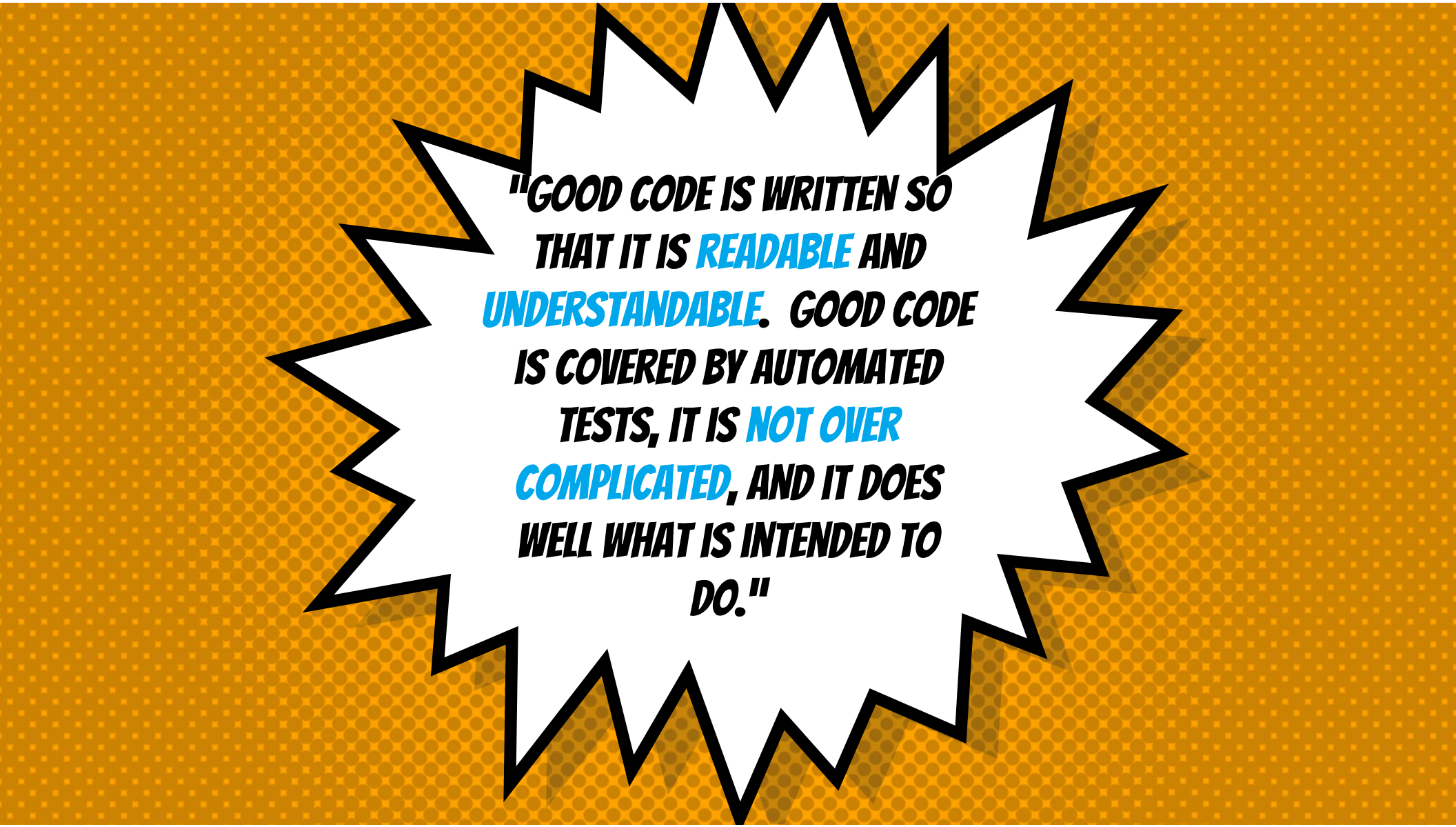
https://www.intenthq.com/good-code-scientific-definition/

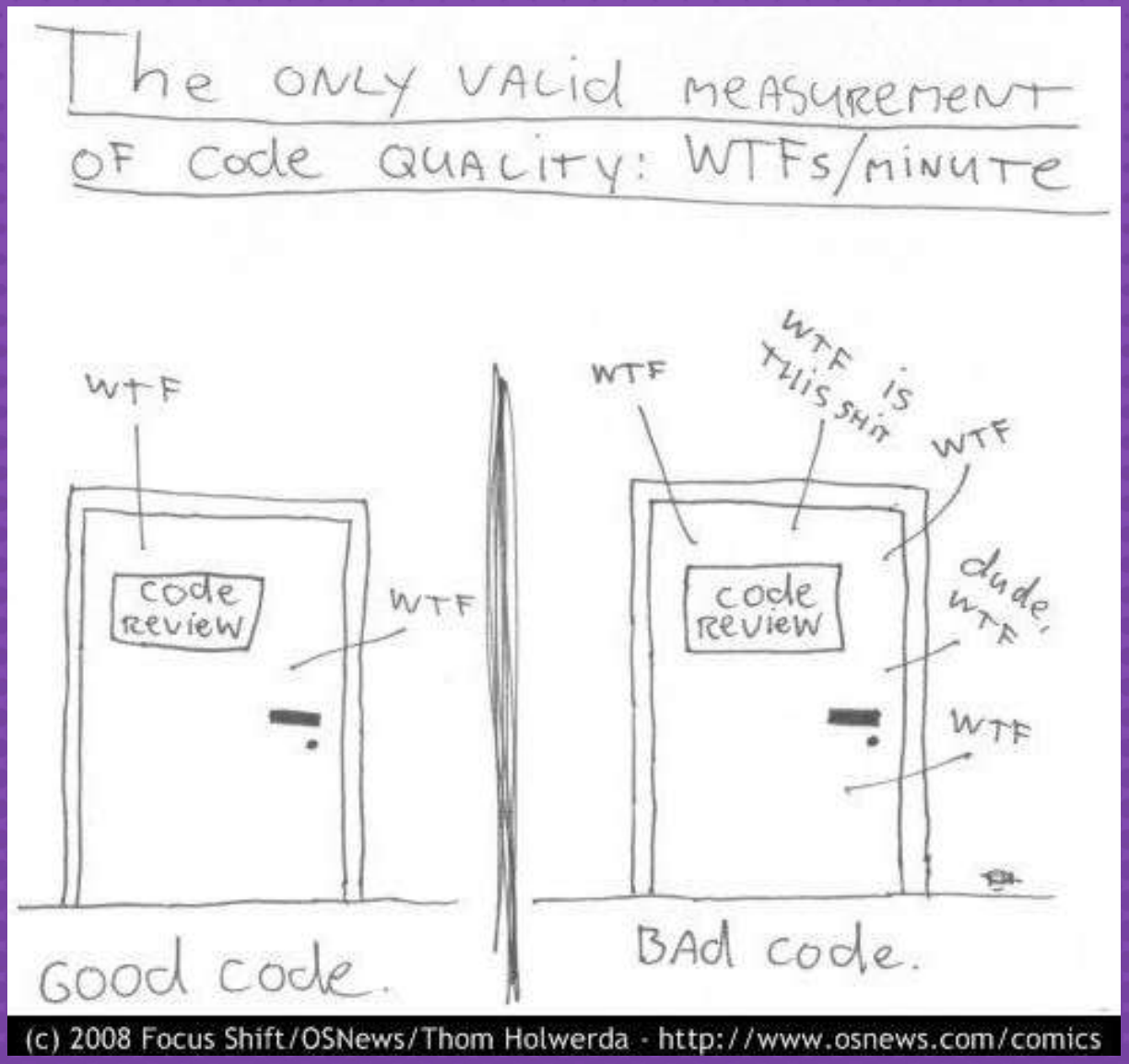

Program. to interf. 3.08%
Object Oriented 3.08%
Few dependencies 3.08%
Comp. over inher. 3.08%
SOLID 4.62%
Robust 4.62%
No prema. optim. 4.62%
Formatted 4.62%
Business 4.62%
Best practices 4.62%
Conventions 6.15%
Concise 6.15%
Clear 6.15%
Structured 9.23%
Single responsibility 9.23%
Patterns 9.23%
Modifiable 9.23%
Small methods 10.77%
Decoupled 10.77%
Naming 12.31%
Efficient 12.31%
DRY 13.85%
Clean 15.38%
Self documented 16.92%
Extendable/Reusable 18.46%
Documented 18.46%
Maintainable 21.54%
Works 24.62%
Simple 24.62%
Testable/Tested 29.23%
Readable 78.46%
0.00%
10.00%
20.00%
30.00%
40.00%
50.00%
60.00%
70.00%
80.00%

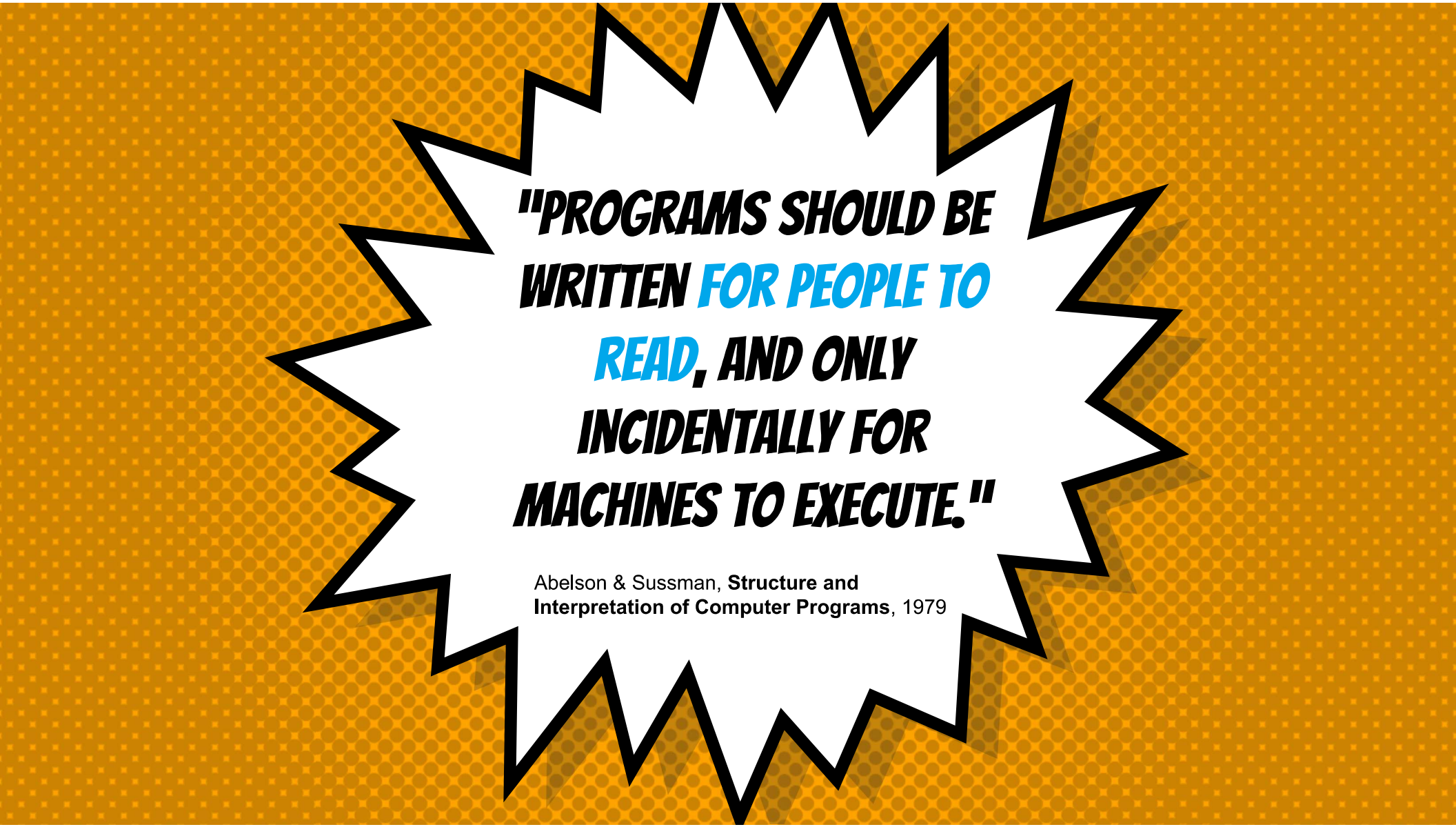
WHAT IS GOOD CODE? A SCIENTIFIC DEFINITION



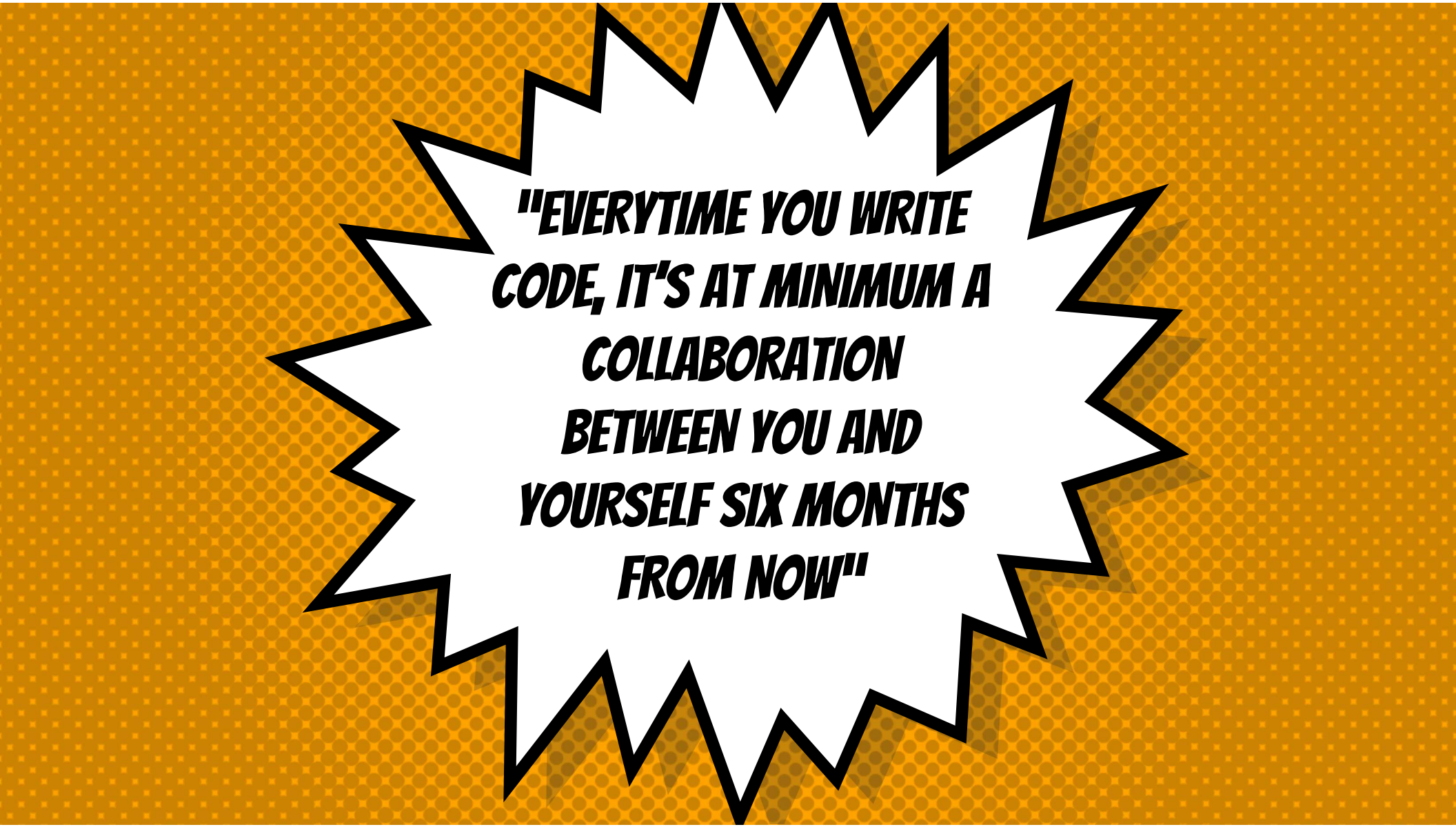
https://www.intenthq.com/good-code-scientific-definition/


"GOOD CODE IS WRITTEN SO THAT IT IS READABLE AND UNDERSTANDABLE. GOOD CODE IS COVERED BY AUTOMATED TESTS, IT IS NOT OVER COMPLICATED, AND IT DOES WELL WHAT IS INTENDED TO DO."


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics


"PROGRAMS SHOULD BE WRITTEN FOR PEOPLE TO READ, AND ONLY INCIDENTALLY FOR MACHINES TO EXECUTE."
Abelson & Sussman, **Structure and Interpretation of Computer Programs**, 1979

BUT NO ONE WILL READ
THIS BUT ME. SO WHY?


"EVERYTIME YOU WRITE CODE, IT'S AT MINIMUM A COLLABORATION BETWEEN YOU AND YOURSELF SIX MONTHS FROM NOW"

OK, BUT HOW DO I BECOME READABLE?

```
var <- TRUE
while(var){
    if(!.overlap(x1-.5*wid,y1-.5*ht,wid,ht,boxes) && x1-.5*wid>xlim[1] && y1-.5*ht>ylim[1] && x1+.5*wid<xlim[2] && y1+.5*ht<ylim[2]){
        boxes[[length(boxes)+1]] <- c(x1-.5*wid,y1-.5*ht,wid,ht)
        var <- FALSE
    }else{
        theta <- theta+tstep
        r <- r + rstep*tstep/(2*pi)
        x1 <- xo+sdx*r*cos(theta); y1 <- yo+sdy*r*sin(theta)
    }}}
```

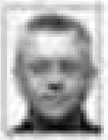
TIP 1

USE SPACING AND LINE LENGTH

OPINION

A4

Beating the heat on family bike trip



```
var <- TRUE
while(var){
    if(!.overlap(x1-.5*wid,y1-.5*ht,wid,ht,boxes) && x1-.5*wid>xlim[1] && y1-.5*ht>ylim[1] && x1+.5*wid<xlim[2] && y1+.5*ht<ylim[2]){
        boxes[[length(boxes)+1]] <- c(x1-.5*wid,y1-.5*ht,wid,ht)
        var <- FALSE
    }else{
        theta <- theta+tstep
        r <- r + rstep*tstep/(2*pi)
        x1 <- xo+sdx*r*cos(theta); y1 <- yo+sdy*r*sin(theta)
    }}}
```

```
var <- TRUE
while(var){
    if(!.overlap(x1-.5*wid,y1-.5*ht,wid,ht,boxes) &&
        x1-.5*wid>xlim[1] &&
        y1-.5*ht>ylim[1] &&
        x1+.5*wid<xlim[2] &&
        y1+.5*ht<ylim[2]){
            boxes[[length(boxes)+1]] <- c(x1-.5*wid,y1-.5*ht,wid,ht)
            var <- FALSE
    }else{
        theta <- theta+tstep
        r <- r + rstep*tstep/(2*pi)
        x1 <- xo+sdx*r*cos(theta); y1 <- yo+sdy*r*sin(theta)
    }}}
```

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid, y1 - .5 * ht, wid, ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r <- r + rstep * tstep / (2 * pi)
        x1 <- xo + sdx * r * cos(theta)
        y1 <- yo + sdy * r * sin(theta)
    }
}
```

TIP 2

ALIGN STUFF

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid, y1 - .5 * ht, wid, ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r <- r + rstep * tstep / (2 * pi)
        x1 <- xo + sdx * r * cos(theta)
        y1 <- yo + sdy * r * sin(theta)
    }
}
```

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht  > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid,
                                            y1 - .5 * ht,
                                            wid,
                                            ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r     <- r + rstep * tstep / (2 * pi)
        x1    <- xo + sdx * r * cos(theta)
        y1    <- yo + sdy * r * sin(theta)
    }
}
```

TIP 3

USE DESCRIPTIVE VARIABLE NAMES

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht  > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid,
                                            y1 - .5 * ht,
                                            wid,
                                            ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r     <- r + rstep * tstep / (2 * pi)
        x1    <- xo + sdx * r * cos(theta)
        y1    <- yo + sdy * r * sin(theta)
    }
}
```

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht  > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid,
                                            y1 - .5 * ht,
                                            wid,
                                            ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r     <- r + rstep * tstep / (2 * pi)
        x1    <- xo + sdx * r * cos(theta)
        y1    <- yo + sdy * r * sin(theta)
    }
}
```

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht  > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid,
                                            y1 - .5 * ht,
                                            wid,
                                            ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r     <- r + rstep * tstep / (2 * pi)
        x1    <- xo + sdx * r * cos(theta)
        y1    <- yo + sdy * r * sin(theta)
    }
}
```

```
isOverlaped <- TRUE
while(isOverlaped) {
    if(!.overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) &&
        x1 - .5 * width > xlim[1] &&
        y1 - .5 * height  > ylim[1] &&
        x1 + .5 * width < xlim[2] &&
        y1 + .5 * height  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                            y1 - .5 * height,
                                            width,
                                            height)

            isOverlaped <- FALSE
    } else {
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi)
        x1    <- x1 + sdx * r * cos(theta)
        y1    <- y1 + sdy * r * sin(theta)
    }
}
```

WARNING

IF YOUR CODE CONTAINS VARIABLES NAMED `VAR` OR `DATA2` OR `DF3`, I WILL PERSONALLY LIGHT IT ON FIRE.

TIP 4

DOCUMENT!

```
var <- TRUE
while(var) {
    if(!.overlap(x1 - .5 * wid, y1 - .5 * ht, wid, ht, boxes) &&
        x1 - .5 * wid > xlim[1] &&
        y1 - .5 * ht  > ylim[1] &&
        x1 + .5 * wid < xlim[2] &&
        y1 + .5 * ht  < ylim[2]) {
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * wid,
                                            y1 - .5 * ht,
                                            wid,
                                            ht)
            var <- FALSE
    } else {
        theta <- theta + tstep
        r     <- r + rstep * tstep / (2 * pi)
        x1    <- xo + sdx * r * cos(theta)
        y1    <- yo + sdy * r * sin(theta)
    }
}
```

```
# Iterate over the boxes and if any boxes are overlapped, adjust them so they are not.
# `x1` and `y1` are the box positions, `xlim` and `ylim` are the dimensions of the
# graphic, `height` and `width` are the height and width of each box.
# `boxes` is a list containing all the metadata.
isOverlaped <- TRUE
while(isOverlaped) {
    if(!.overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) &&
        x1 - .5 * width > xlim[1] &&
        y1 - .5 * height  > ylim[1] &&
        x1 + .5 * width < xlim[2] &&
        y1 + .5 * height  < ylim[2]) {
            # Record corrected dimensions in box metadata
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                            y1 - .5 * height,
                                            width,
                                            height)
            isOverlaped <- FALSE
    } else {
        # Keep trying to adjust the dimensions until we are not overlapped
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi) # Change the rotation of box
        x1    <- x1 + sdx * r * cos(theta)        # Change the x position
        y1    <- y1 + sdy * r * sin(theta)        # Change the y position
    }
}
```

TIP 5

USE HELPER FUNCTIONS

```
# Iterate over the boxes and if any boxes are overlapped, adjust them so they are not.
# `x1` and `y1` are the box positions, `xlim` and `ylim` are the dimensions of the
# graphic, `height` and `width` are the height and width of each box.
# `boxes` is a list containing all the metadata.
isOverlaped <- TRUE
while(isOverlaped) {
    if(!.overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) &&
        x1 - .5 * width > xlim[1] &&
        y1 - .5 * height  > ylim[1] &&
        x1 + .5 * width < xlim[2] &&
        y1 + .5 * height  < ylim[2]) {
            # Record corrected dimensions in box metadata
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                            y1 - .5 * height,
                                            width,
                                            height)
            isOverlaped <- FALSE
    } else {
        # Keep trying to adjust the dimensions until we are not overlapped
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi) # Change the rotation of box
        x1    <- x1 + sdx * r * cos(theta)        # Change the x position
        y1    <- y1 + sdy * r * sin(theta)        # Change the y position
    }
}
```

```r
# Iterate over the boxes and if any boxes are overlapped, adjust them so they are not.
# `x1` and `y1` are the box positions, `xlim` and `ylim` are the dimensions of the
# graphic, `height` and `width` are the height and width of each box.
# `boxes` is a list containing all the metadata.
isOverlaped <- TRUE
while(isOverlaped) {
    if(!.overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) &&
        x1 - .5 * width > xlim[1] &&
        y1 - .5 * height  > ylim[1] &&
        x1 + .5 * width < xlim[2] &&
        y1 + .5 * height  < ylim[2]) {
            # Record corrected dimensions in box metadata
            boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                            y1 - .5 * height,
                                            width,
                                            height)
            isOverlaped <- FALSE
    } else {
        # Keep trying to adjust the dimensions until we are not overlapped
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi) # Change the rotation of box
        x1    <- x1 + sdx * r * cos(theta)        # Change the x position
        y1    <- y1 + sdy * r * sin(theta)        # Change the y position
    }
}
```

```r
IsOverlaped <- function(x1, y1, width, height, boxes) {
    # Use internal C .overlap function to determine overlap in graphics,
    # also check the boundaries of the graphic contained in `xlim`
    .overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) ||
        x1 - .5 * width <= xlim[1] ||
        y1 - .5 * height  <= ylim[1] ||
        x1 + .5 * width > xlim[2] ||
        y1 + .5 * height  > ylim[2]
}

# Iterate over the boxes and if any boxes are overlapped, adjust them so they are not.
# `x1` and `y1` are the box positions, `xlim` and `ylim` are the dimensions of the
# graphic, `height` and `width` are the height and width of each box.
# `boxes` is a list containing all the metadata.
isOverlaped <- TRUE
while(isOverlaped) {
    if(!IsOverlapped(x1, y1, width, height, boxes) {
        # Record corrected dimensions in box metadata
        boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                        y1 - .5 * height,
                                        width,
                                        height)
        isOverlaped <- FALSE
    } else {
        # Keep trying to adjust the dimensions until we are not overlapped
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi) # Change the rotation of box
        x1    <- x1 + sdx * r * cos(theta)        # Change the x position
        y1    <- y1 + sdy * r * sin(theta)        # Change the y position
    }
}
```

TIP 6

AVOID INTERMEDIARY ASSIGNMENT

```r
# If we list all the natural numbers below 10 that are multiples of 3 or 5,
# we get 3, 5, 6 and 9. The sum of these multiples is 23.

# Find the sum of all the multiples of 3 or 5 below 1000.
numbers_below_1000 <- seq(999)
multiples_of_three_below_1000 <- c()
for (i in numbers_below_1000) {
    if (i %% 3 == 0) {
        multiples_of_three_below_1000 <- append(multiples_of_three_below_1000, i)
    }
}
multiples_of_3_and_5_below_1000 <- c()
for (i in multiples_of_three_below_1000) {
    if (i %% 5 == 0) {
        multiples_of_3_and_5_below_1000 <- append(multiples_of_3_and_5_below_1000, i)
    }
}
answer <- sum(multiples_of_3_and_5_below_1000)
```

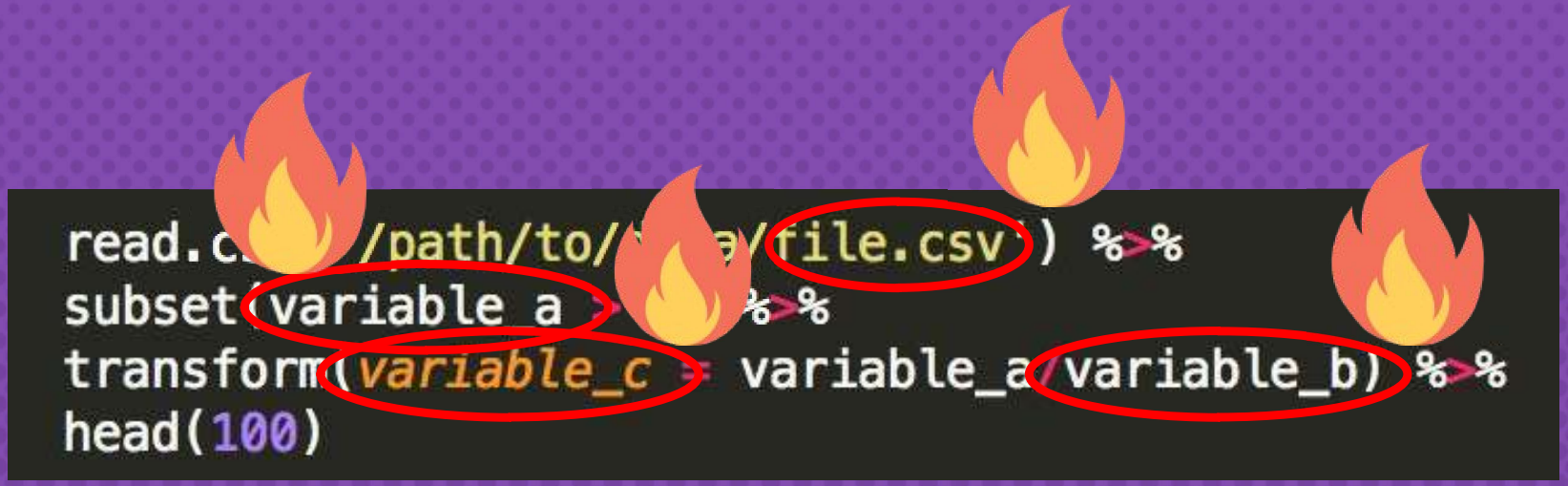
```r
# If we list all the natural numbers below 10 that are multiples of 3 or 5,
# we get 3, 5, 6 and 9. The sum of these multiples is 23.

# Find the sum of all the multiples of 3 or 5 below 1000.
multiples_of_3_and_5_below_1000 <- c()
for (i in seq(999)) {
    if (i %% 5 == 0 || i %%3 == 0) {
        multiples_of_3_and_5_below_1000 <- append(multiples_of_3_and_5_below_1000, i)
    }
}
answer <- sum(multiples_of_3_and_5_below_1000)
```

TIP 7

PIPING WITH MAGRITTR IS AWESOME

```
head(transform(subset(read.csv('/path/to/data/file.csv'),
    variable_a > x),
    variable_c = variable_a / variable_b),
    100)
```

```
read.csv('/path/to/data/file.csv') %>%
subset(variable_a > x) %>%
transform(variable_c = variable_a/variable_b) %>%
head(100)
```


read.c /path/to/ file.csv) %>%
subset(variable_a > %>%
transform(variable_c = variable_a / variable_b) %>%
head(100)

TIP 8

USE FUNCTIONAL PROGRAMMING

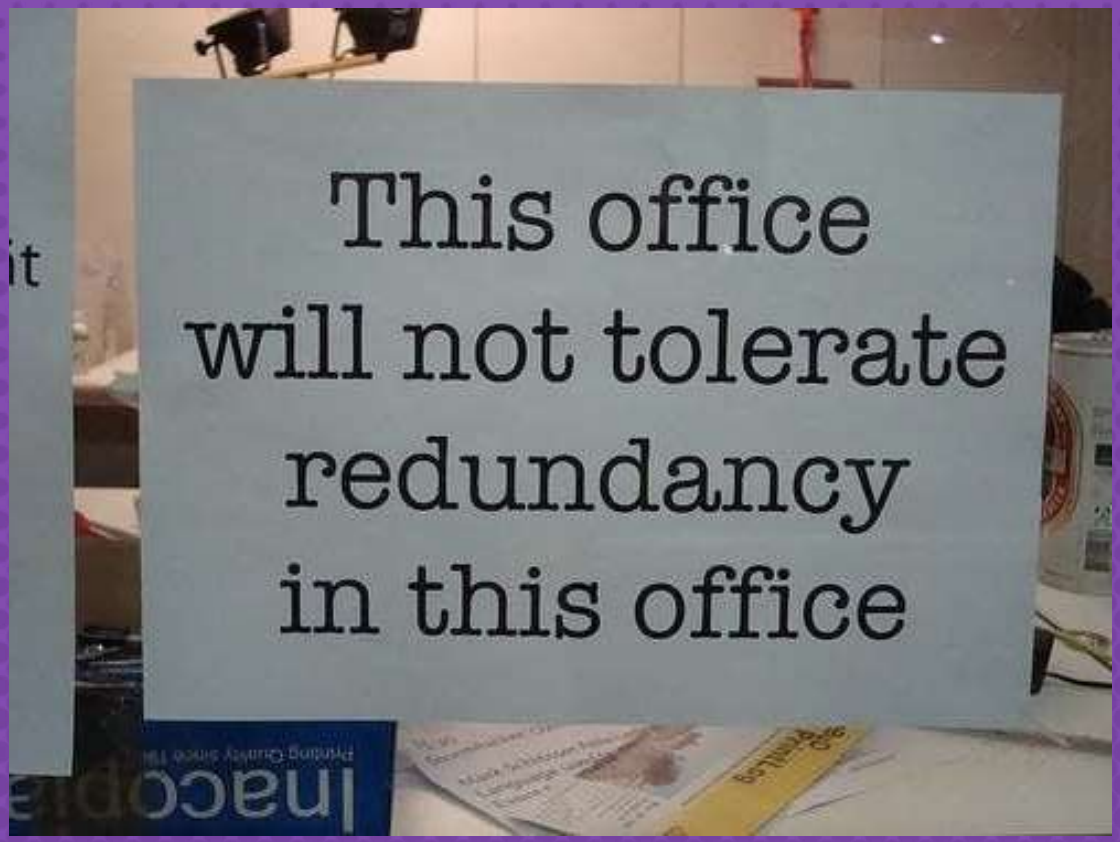
```r
# If we list all the natural numbers below 10 that are multiples of 3 or 5,
# we get 3, 5, 6 and 9. The sum of these multiples is 23.

# Find the sum of all the multiples of 3 or 5 below 1000.
multiples_of_3_and_5_below_1000 <- c()
for (i in seq(999)) {
    if (i %% 5 == 0 || i %%3 == 0) {
        multiples_of_3_and_5_below_1000 <- append(multiples_of_3_and_5_below_1000, i)
    }
}
answer <- sum(multiples_of_3_and_5_below_1000)
```

```r
# If we list all the natural numbers below 10 that are multiples of 3 or 5,
# we get 3, 5, 6 and 9. The sum of these multiples is 23.

# Find the sum of all the multiples of 3 or 5 below 1000.
answer <- sum(Filter(function(x) { x %% 3 == 0 || x %% 5 == 0 }, seq(999)))
```

```r
# If we list all the natural numbers below 10 that are multiples of 3 or 5,
# we get 3, 5, 6 and 9. The sum of these multiples is 23.

# Find the sum of all the multiples of 3 or 5 below 1000.
answer <- seq(999) %>%
            Filter(function(x) { x %% 3 == 0 || x %% 5 == 0 }) %>%
            sum
```

TIP 9

DON'T REPEAT YOURSELF


This office
will not tolerate
redundancy
in this office

```
z1 <- x1 + 365 - f(y1)
z2 <- x2 + 365 - f(y2)
z3 <- x3 + 365 - f(y3)
```

```
z <- lapply(x, x + 365 - f(y))
```

TIP 10

CREATE AND FOLLOW A STYLE GUIDE

STYLE.TIDYVERSE.ORG

The tidyverse style guide

Hadley Wickham

Welcome

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. This site describes the style used throughout the tidyverse. It was originally derived from Google's R style guide, but has evolved and expanded considerably over the years.

TIP 11

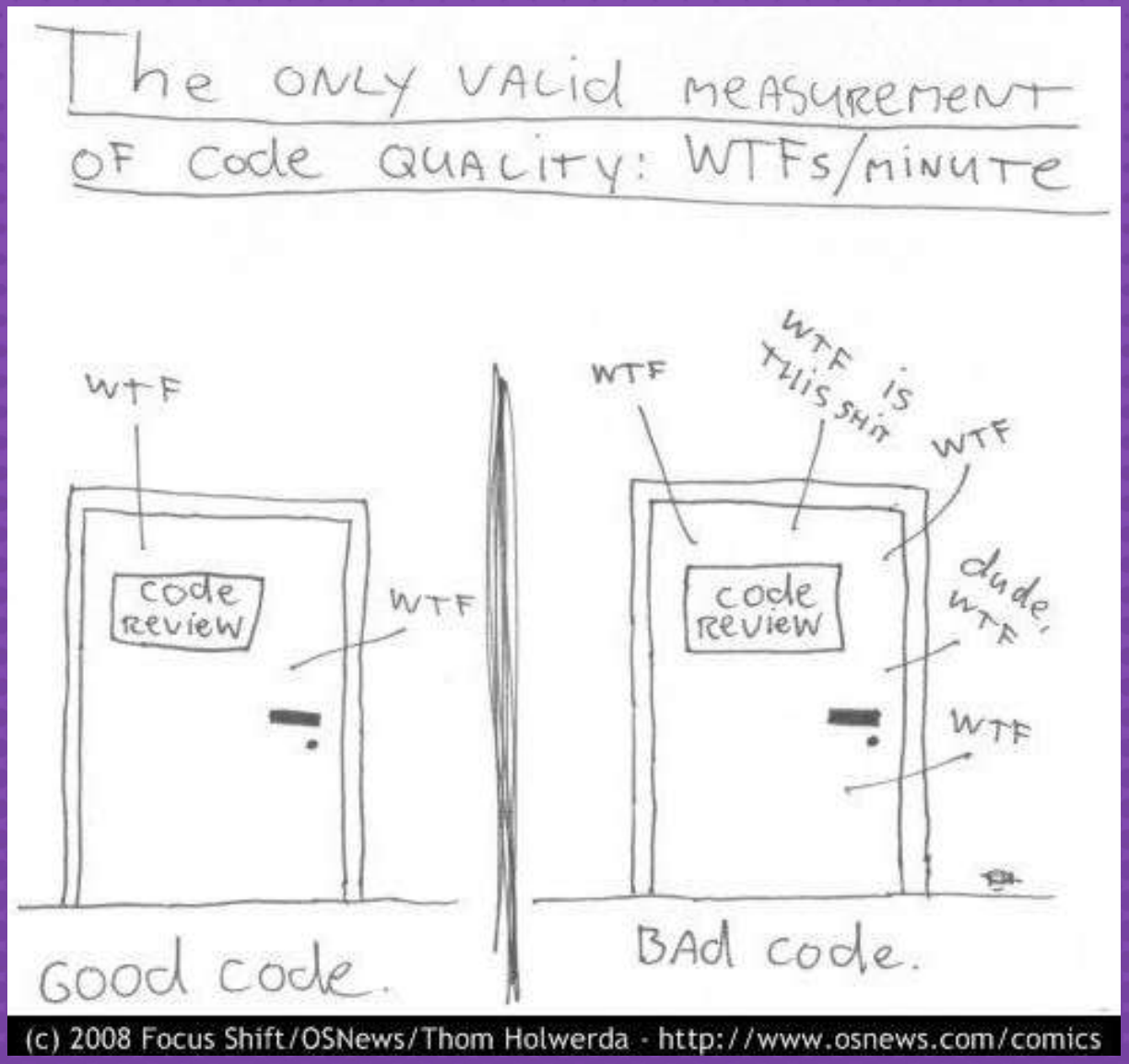
USE TESTS TO MAKE SURE YOUR REFACTORING DOESN'T BREAK STUFF

TIP 12

ULTIMATE TEST: HAVE SOMEONE ELSE TELL YOU WHAT YOUR CODE DOES

RECAP, PLEASE?

1. Use spacing and line length
2. Align code vertically
3. Use descriptive variable names
4. Document your code
5. Use helper functions to chunk code
6. Avoid intermediary assignment
7. Piping with magrittr is awesome
8. Use functional programming (`Filter`, `lapply`)
9. Don't repeat yourself
10. Create and follow a style guide
11. Use tests to make sure your refactoring doesn't break stuff
12. Have someone tell you what your code does


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

```
var <- TRUE
while(var){
    if(!.overlap(x1-.5*wid,y1-.5*ht,wid,ht,boxes) && x1-.5*wid>xlim[1] && y1-.5*ht>ylim[1] && x1+.5*wid<xlim[2] && y1+.5*ht<ylim[2]){
        boxes[[length(boxes)+1]] <- c(x1-.5*wid,y1-.5*ht,wid,ht)
        var <- FALSE
    }else{
        theta <- theta+tstep
        r <- r + rstep*tstep/(2*pi)
        x1 <- xo+sdx*r*cos(theta); y1 <- yo+sdy*r*sin(theta)
    }}}
```

```r
IsOverlaped <- function(x1, y1, width, height, boxes) {
    # Use internal C .overlap function to determine overlap in graphics,
    # also check the boundaries of the graphic contained in `xlim`
    .overlap(x1 - .5 * width, y1 - .5 * height, width, height, boxes) ||
        x1 - .5 * width <= xlim[1] ||
        y1 - .5 * height  <= ylim[1] ||
        x1 + .5 * width > xlim[2] ||
        y1 + .5 * height  > ylim[2]
}

# Iterate over the boxes and if any boxes are overlapped, adjust them so they are not.
# `x1` and `y1` are the box positions, `xlim` and `ylim` are the dimensions of the
# graphic, `height` and `width` are the height and width of each box.
# `boxes` is a list containing all the metadata.
isOverlaped <- TRUE
while(isOverlaped) {
    if(!IsOverlapped(x1, y1, width, height, boxes) {
        # Record corrected dimensions in box metadata
        boxes[[length(boxes) + 1]] <- c(x1 - .5 * width,
                                        y1 - .5 * height,
                                        width,
                                        height)
        isOverlaped <- FALSE
    } else {
        # Keep trying to adjust the dimensions until we are not overlapped
        theta <- theta + thetaStep
        r     <- r + rstep * thetaStep / (2 * pi) # Change the rotation of box
        x1    <- x1 + sdx * r * cos(theta)        # Change the x position
        y1    <- y1 + sdy * r * sin(theta)        # Change the y position
    }
}
```

1. Use spacing and line length
2. Align code vertically
3. Use descriptive variable names
4. Document your code
5. Use helper functions to chunk code
6. Avoid intermediary assignment
7. Piping with magrittr is awesome
8. Use functional programming (`Filter`, `lapply`)
9. Don't repeat yourself
10. Create and follow a style guide
11. Use tests to make sure your refactoring doesn't break stuff
12. Have someone tell you what your code does