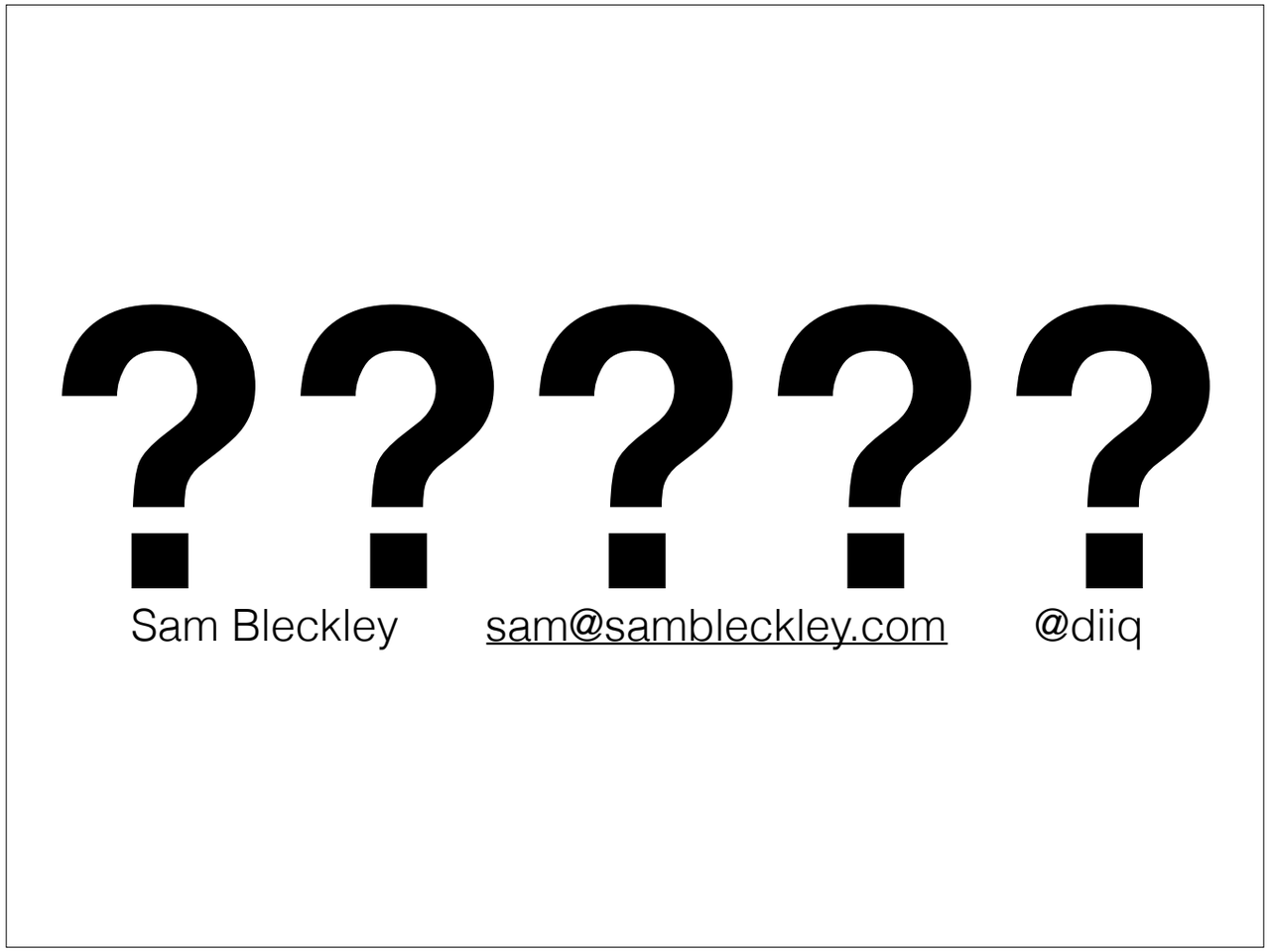


Hi, I'm Sam Bleckley. I am a software engineer, designer, and consultant.

I have a cold, so please forgive me if I run out the door as soon as this presentation is over — it's for your own good! If you have questions, or want to start a conversation, please email me or tweet at me. I'll leave some business cards up here.

who are you people

what are you doing here

why aren't you someplace nice

Why do you want to learn to code? Raise your hand if

- I have a cool project I wanna do
- I wanna get a job
- It’s a hobby, like playing the saxophone or making buildings out of matchsticks
- I know lots of programmers, and I want to understand them better
- I am already a professional coder, I don’t need your stinkin’ advice

I wanna talk about the craft of writing code — how to make beautiful, clear, joyful code. It's not what will impress on a resume, but it *will* bring you the praise and friendship of any other software developers you work with.

It may seem premature to talk about beautiful code when many of you are still learning to write code that works at all

But you're at a key point in your lives as writers of code — these early hours and days of repetitive coding are incredibly important

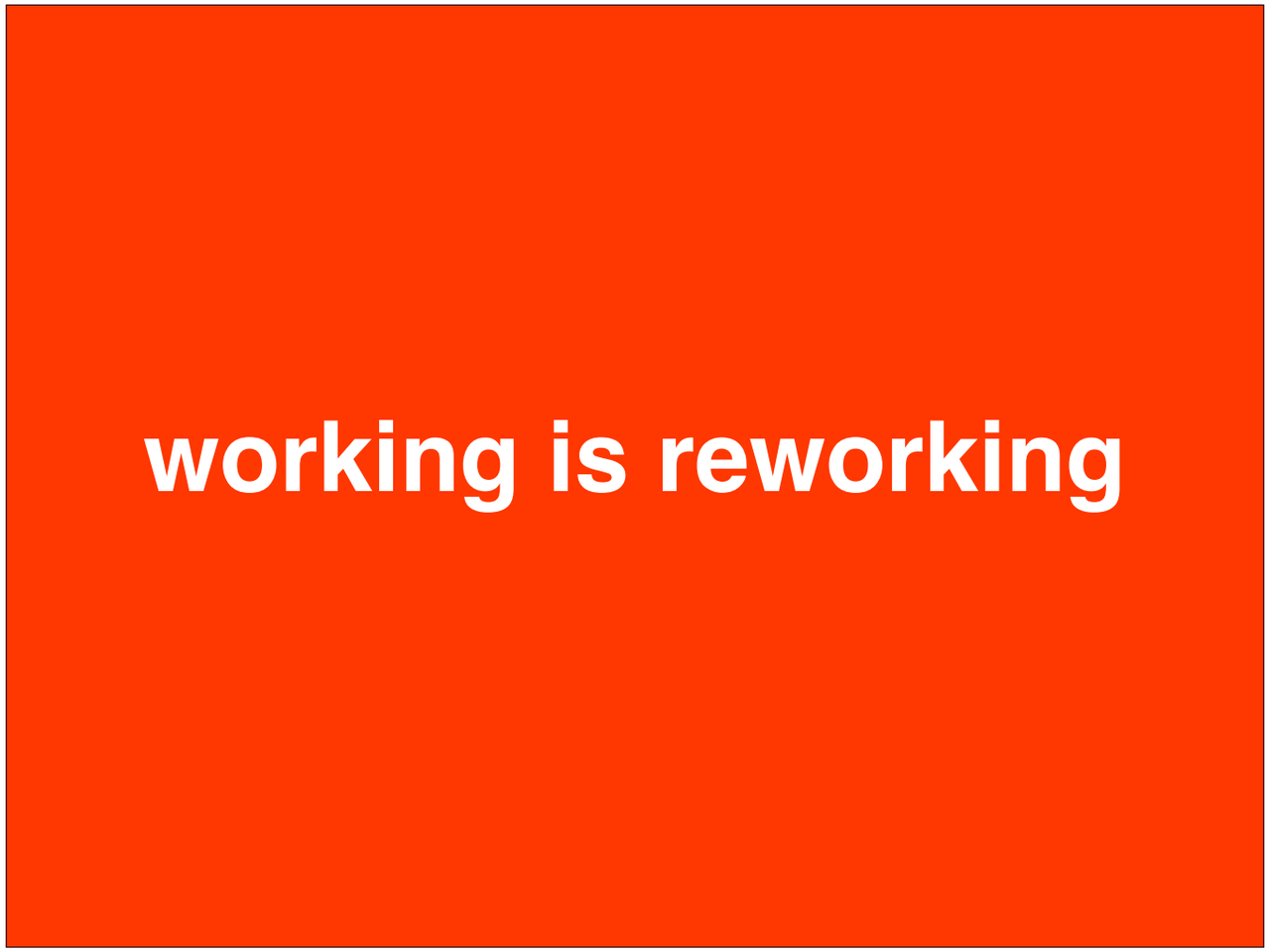
repetition is how you say

“this is the way I want to do this”

(but faster and with less effort)

If you want to ever write beautiful code, you have to rehearse writing beautiful code even when it doesn’t matter.

But there’s good news! You don’t have to produce beautiful perfect code straight from your mind through your hands to your keyboard!

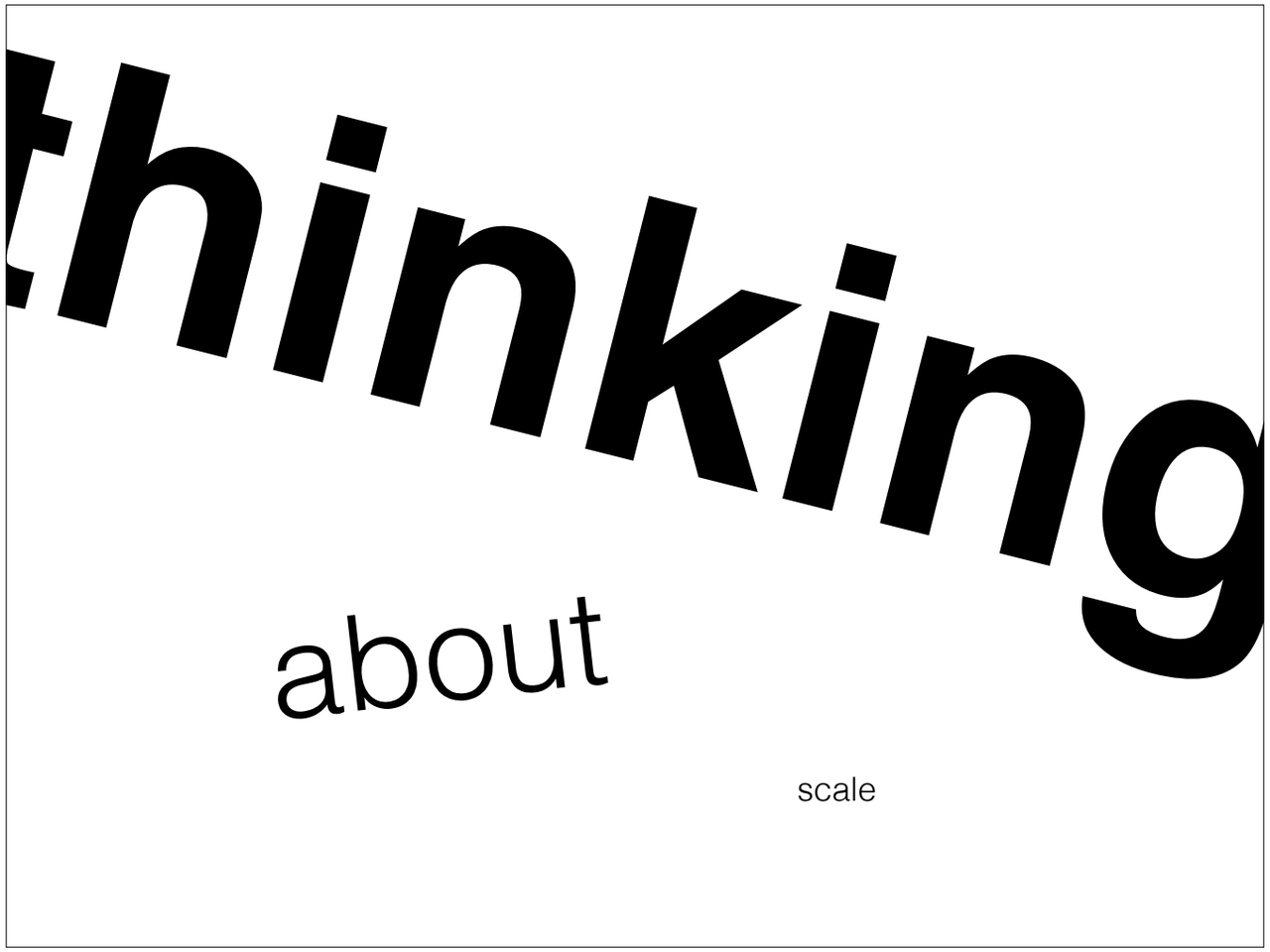


All working is reworking! All working is reworking!

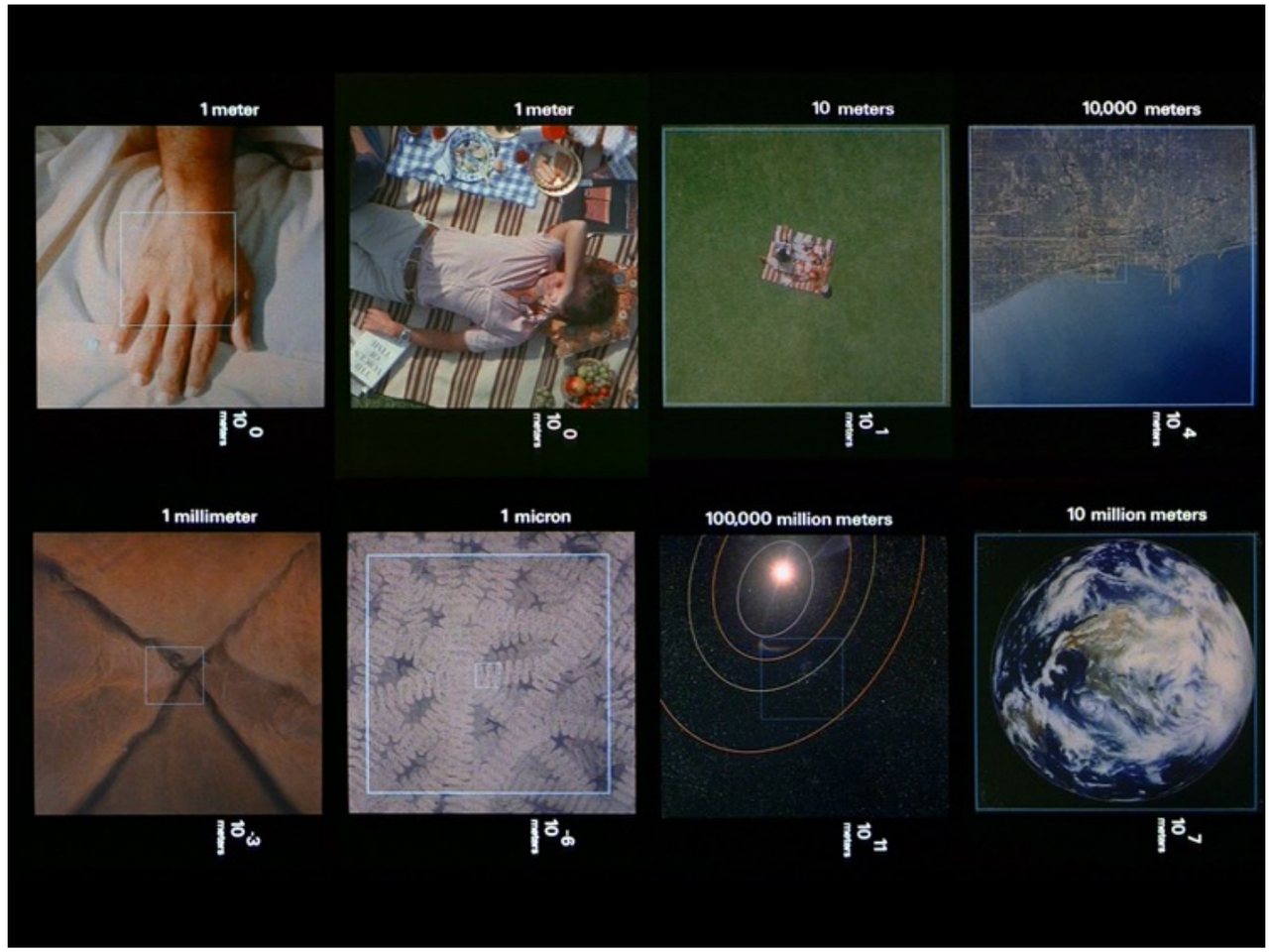
Get your ideas on the page as fast as possible; because then you don't need a framework for creation, you just need to be able to incrementally improve it.

A blank canvas is terrifying; but a sketchy bit of code that has problems can slowly be made into a sketchy bit of code with fewer problems!

OK. Let’s get some conceptual groundwork done.



Lets think about scale.



How many people recognize this?

These are frames from the Eames' Powers of Ten. It's a 10 minute video, and I highly encourage you to seek it out if it's unfamiliar. It looks are the universe through a successively larger window, and then through a successively smaller one.

What does that mean for code? What's the smallest unit of code that's still code?

What's the largest unit?

line

function / method / procedure / definition

class / module / file

program / application / library

suite / stack / toolchain

Here’s a vague sort of hierarchy I came up with, working through successively larger scales.
We’re going to talk about techniques that work all the way up and all the way down this list; keep that in mind, and in your head be ready to swap between thinking about individual lines of code, classes, and whole programs.

[if there's time define "scale invariance" but it's not necessary]

line

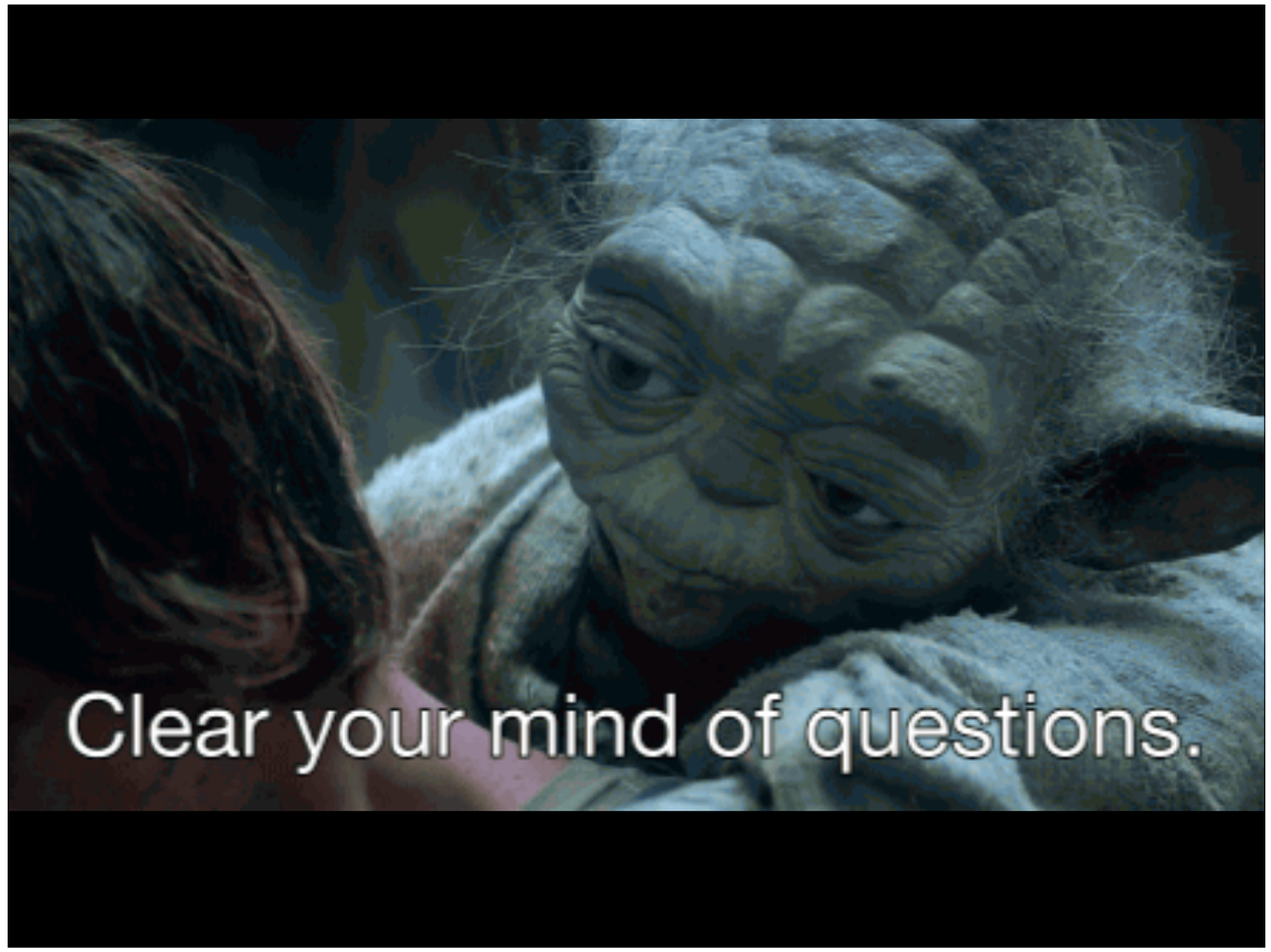
function

class

program

For simplicity, I’m going to try to stick to using these words for each scale. Different languages and paradigms will use different words, but I'm talking about the same thing.

Any questions?



Now, clear your mind of questions.

wait

Wait, no.

Don't clear your mind of questions.
I meant **don't**.

Questions are amazing.

Questions are a good way of finding things out.

They're also a great way to prevent yourself from doing lipservice. It's very easy to say "I believe in writing code that's simple and easy to read" — and nevertheless write whatever the heck code you want, anyway. If you make a habit to ask yourself "what part of this code is the hardest to read?" and always force yourself to answer, then you're always aware of the weakest parts of what you're writing.

(Go story here depending on time, energy, engagement)

1. What **question** does it **answer**?

2. What **action** does it **take**?

So here are our first two questions.

Any line, any function, any class, any program, should aim to answer exactly one or exactly zero questions.

Any line, any function, any class, any program, should aim to take exactly zero or exactly one actions.

What the heck does that mean?

Calendar.app
Gmail
Mint
Twitter
Square

What one question do these programs answer? What one action do each of these programs take?

Array#sort
print
Time.now()
fetch_cat_picture()

Ditto, here are some function names.

```c
token *read_open(FILE *stream){
    int c = fgetc(stream);
    if(c != '('){
        ungetc(c, stream);
        return NULL;
    }
    return make_token(OPEN_TOKEN);
}
```

```python
    def complete(self):
        return self.sessions_completed == self.experiment.session_count
```

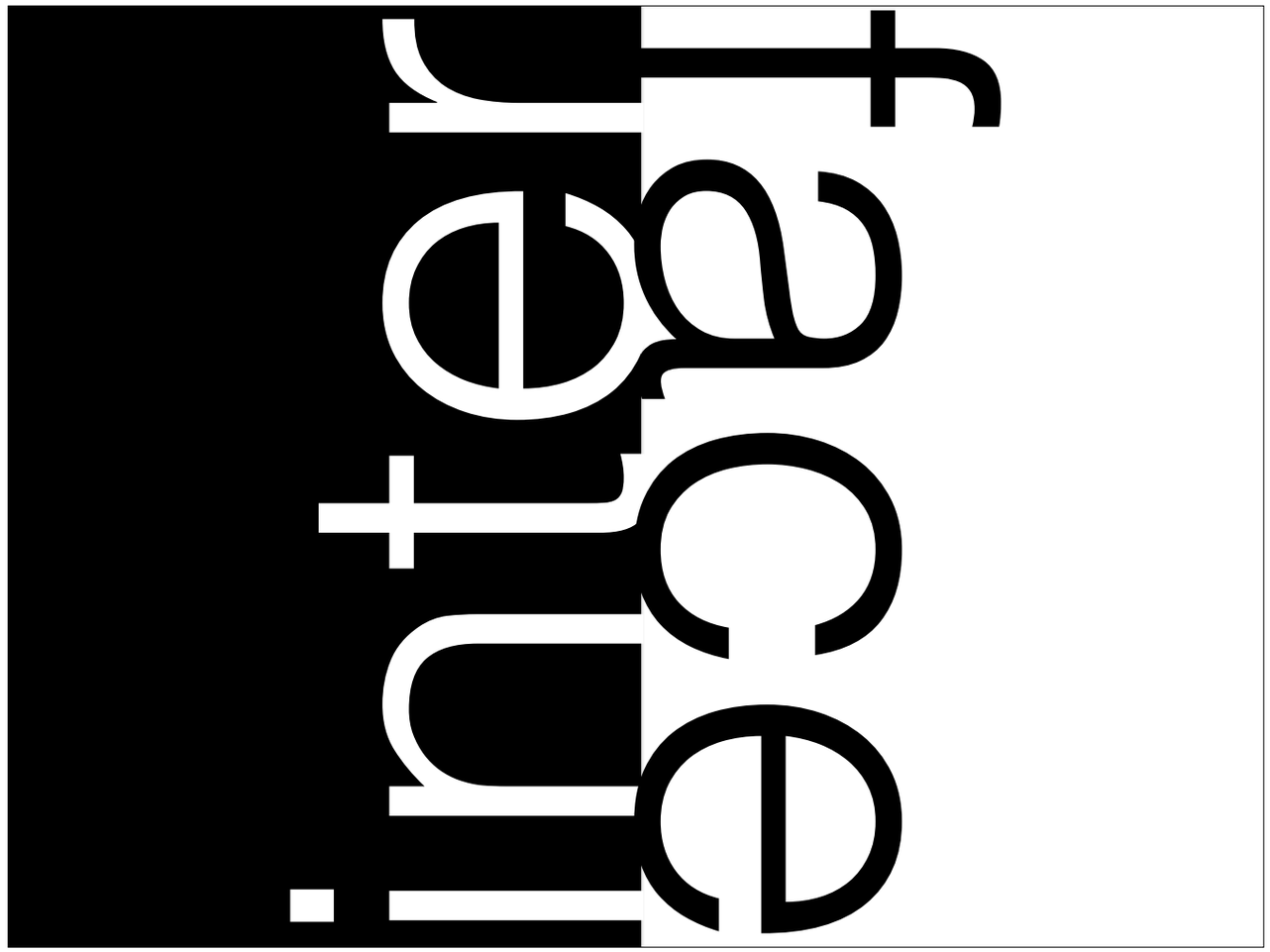
```
# The Game loop
setInterval ->
  thisStep = now()
  steps = thisStep - lastStep

  # If you change tabs, steps can be a bajillion. Don't let that
  # happen.
  if steps > 500
    lastStep = thisStep
    return

  # move the world
  world.moveStuff(steps)

  # move the player
  world.player.move(steps, world.blinds)

  dead = world.player.updateHealth(
    history.lightHistory.evilPolys(),
    [world.light.litPolygon(world.blinds)],
    steps
  )

  if dead
    setWorld(worlds.dead)
    setTimeout ->
      if worldInd == worlds.dead
        setWorld(lastWorldInd)
    , 7000

  if world.goal.win(world.player)
    points += Math.round(world.player.health)
    setWorld(worldInd + 1)

  # HAHAHAHAHAHAHAAAAXXXXXXX

  if worldInd == worlds.intro and input.anyKeysDown "space", "enter"
    setWorld(worlds.first)

  if worldInd == worlds.dead and input.anyKeysDown "space", "enter"
    setWorld(lastWorldInd)

  # REDRAW!!

  canvas.clear()
  world.drawBottom(canvas)
  history.drawLightHistory(canvas)
  canvas.add world.player.fabricObject()
  world.drawTop(canvas)

  score = new fabric.Text "Score: #{points}",
    fill: "#7c8"
    top: 580
    left: 500
  score.setFontSize 12
  score.setFontFamily "Courier New"

  canvas.add score
  canvas.renderAll()

  lastStep = thisStep
```

Here’s a terrifically bad example.

design

vs

implementation



By interface, I *don't* mean the visual interface — I am literally talking about the surface area of the line, the function. What does it need in order to run? What does it produce?

These first two questions ("what action does it take", "what question does it answer") are questions about the interface — what goes in and what comes out.

The interface can and should be worked out ahead of time. Don't worry - you can change your mind.

But, because it is part of the plan you work out *before* you write the line, the function, the program, whatever — I'm going to call that "design"

But there are some bits between the — that's implementation.

OK, in order to make short, memorable questions about implementation, I've had to make up some words.

Code is still very new (relative to the rate of change of language) and the population of people who write code is still small (relative to the number of people who talk about tech) so (aside from proper nouns) language lags behind.

I think these are good words that I have made up, but fair warning: if you use them, strangers may thing you're strange.

FIT!

fit is a measure of how well the code
matches the design

A secret: most code is wrong. Even correct code is wrong.

It's usually wrong in ways that don't matter — but think of the Y2K bug! That was code that worked perfectly as long as the domain of years was between 1900 and 1999, and failed outside that range — which, in 1976 is perfectly acceptable.

So even code that is "perfectly fine" doesn't have perfect **fit**.

“it should shuffle a list of integers”

Here’s an even subtler example. I want to write a function that shuffles a list of integers.

```
void shuffle (int* cards, int length) {
  for (int i = 0; i < length; i++) {
    int n = rand_int(length);
    swap(cards, i, n);
  };
}
```

3. In what ways does this **fail** to **fit**?

“it should **fairly** shuffle a list of integers”

Part of thinking about design and about fit, is thinking about the ways your design is *vague* or *underspecified*. What does shuffle mean?

Judging fit requires a deep understanding of the design — the better you understand what’s needed, the better you can compare that to what you have.

```
void better_shuffle (int* cards, int length) {
  for (int i = 0; i < length; i++) {
    int n = rand_int(i + 1);
    swap(cards, i, n);
  };
}
```

Remember — you are allowed to *accept* a lack of fit — but you must do your best to *recognize* it. Asking “where does it fail to fit” forces your hand.

weight

weight is a measure of how hard it is
for someone unfamiliar with this
code to pick it up

```
"Y": {id:"RCL", src:"cclock.png", step : map_swap(
          {"WSL":"WSU",
           "GSL":"GSU",
           "OSL":"OSU",
           "WSU":"WSR",
           "GSU":"GSR",
           "OSU":"OSR",
           "WSR":"WSD",
           "GSR":"GSD",
           "OSR":"OSD",
           "WSD":"WSL",
           "GSD":"GSL",
           "OSD":"OSL"})},

"Z": {id:"RCC", src:"clock.png", step : map_swap(
          {"WSL":"WSD",
           "GSL":"GSD",
           "OSL":"OSD",
           "WSU":"WSL",
           "GSU":"GSL",
           "OSU":"OSL",
           "WSR":"WSU",
           "GSR":"GSU",
           "OSR":"OSU",
           "WSD":"WSR",
           "GSD":"GSR",
           "OSD":"OSR"})},
```

4. Which parts are the **heaviest**?

weight depends on your audience! One rails developer asking about weight with respect to other rails developers is very different than asking about an apprentice programmer, or an experienced embedded programmer.

judging weight requires empathy — the ability to put yourself in some foreign programmer's mindset. The more of other-people's code you read, the better you'll be able to do this.

stretch

stretch is a measure of how easily code will adapt to future changes in design

This is the hardest. Judging stretch, and what kinds of stretch are most useful, takes experience.

“what is the sum of these three numbers?”

So, given this design statement

```python
def add_three_values(a, b, c):
    return a + b + c

# add_three_values(1, 2, 3) => 6
```

Here’s a fn that fits pretty darn closely. So what about stretch? What about the design spec *might* change?

5. Which parts are the **least stretchy**?

```python
def add_up(*numbers):
    return reduce(lambda x, y: x + y, numbers, 0)

# add_up(1, 2, 3) => 6
# add_up(1, 2, 3, 4) => 10
```

This is much heavier — you have to know lots more about python to understand it — but it's much stretchier, too — it can any number of arguments.

```
sum([1, 2, 3, 4])  # => 10
```

Before any python people yell at me, there is a built-in sum function that works like this.

1. What **question** does it **answer**?
2. What **action** does it **take**?
3. In what ways does it **fail** to **fit**?
4. Which parts are the **heaviest**?
5. Which parts are the **least stretchy**?

Sam Bleckley
@diiq
sam@samblecklеy.com

Ask these questions repeatedly, in this order, about your code, and you will always have a list of priorities about what to fix, and a fair amount of confidence that fixing makes things better.