Building Worlds
&
Feeling Great

Brad Fults · @h3h
Head of Engineering Operations
Under Armour

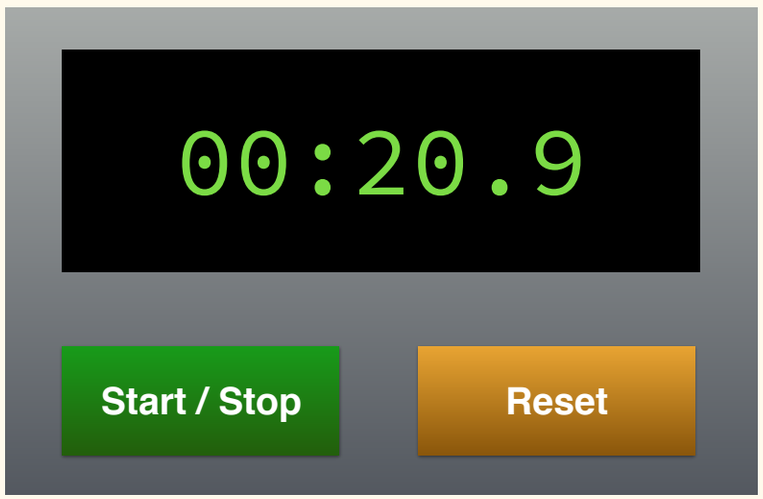
#buildingworlds

Hi, I’m Brad and I work for Under Armour.
I hope you come away from today with a little bit of insight into building worlds and feeling great.
“Building Worlds” is about systems, some systems are software.

Interfaces

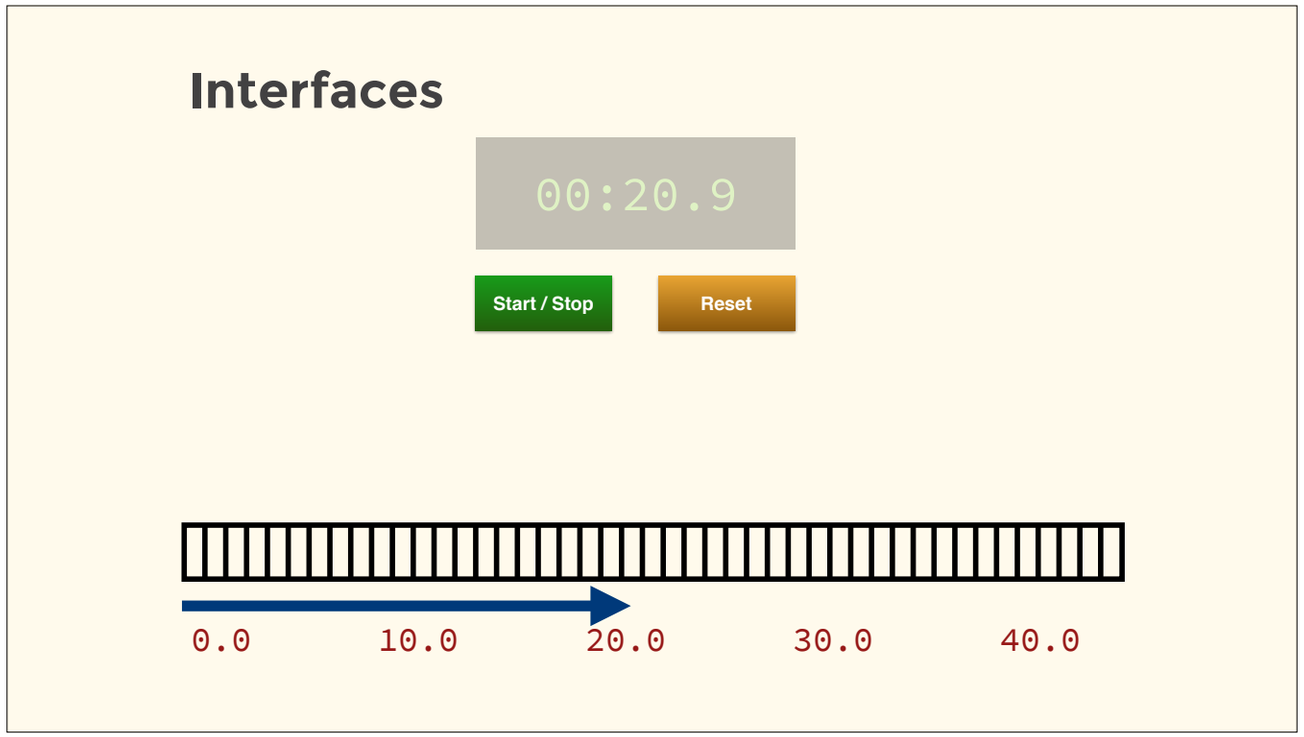
As we’ll soon find out, all systems have interfaces.
I think of interfaces as the edges of tiny little worlds that we build in our minds.
Often, the worlds we build are composed of many other little words, forming a fractal of remarkable function.
But I’m getting ahead of myself, so let’s start at the beginning.

Interfaces

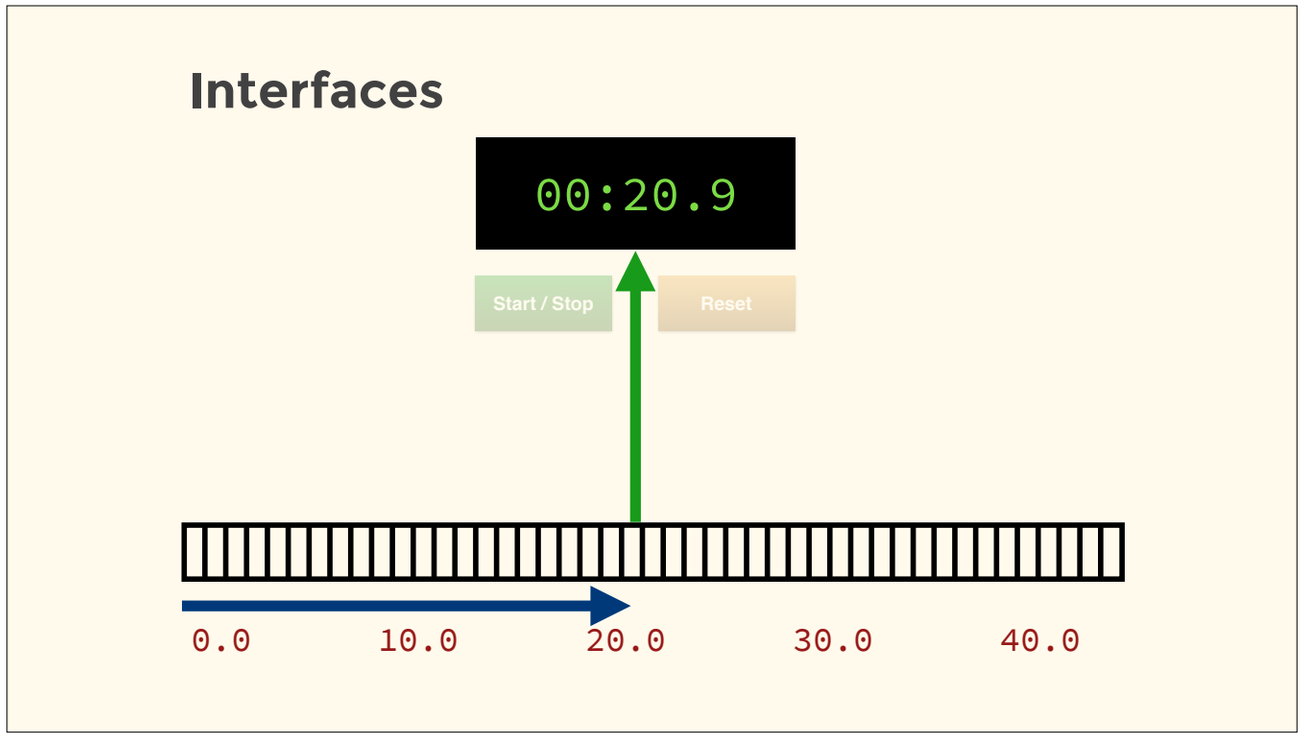


Some interfaces make use of a human’s eyes & fingers, like a stopwatch.

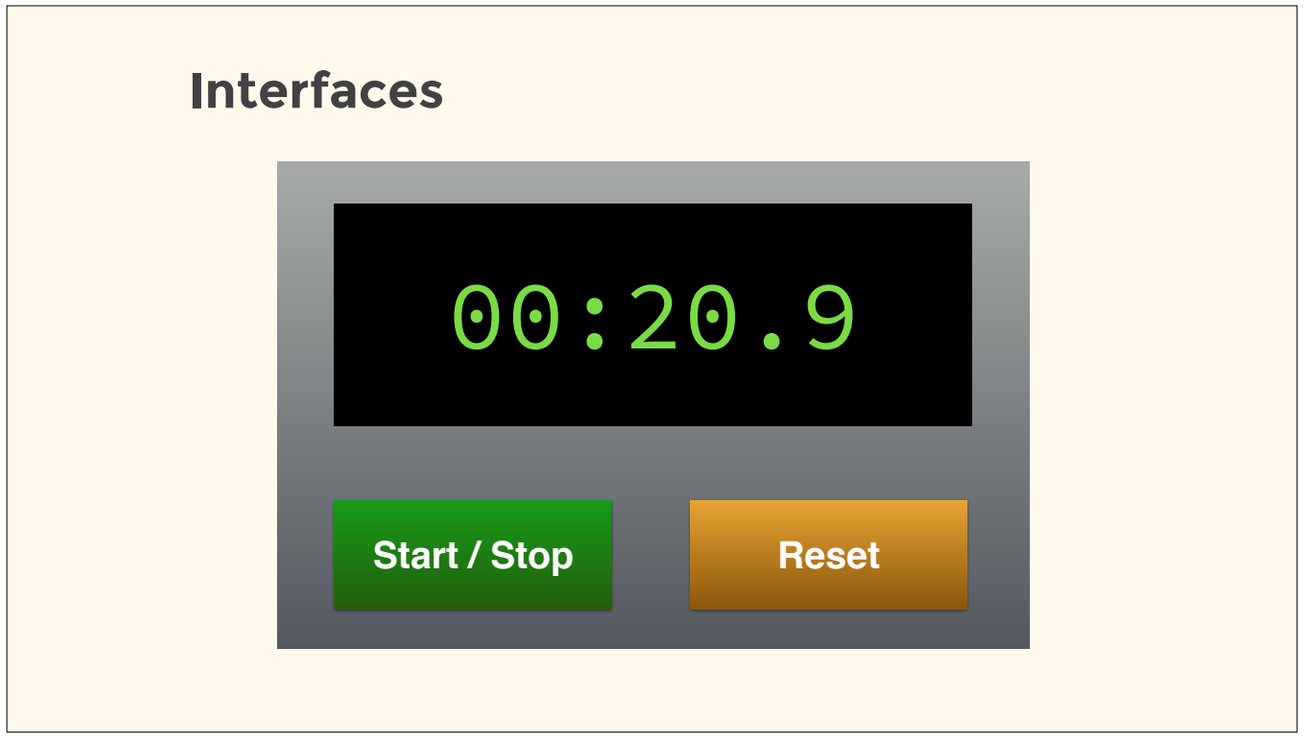
There’s something else going on behind the scenes, though.



I think of it as a timeline inside the system, being controlled by the two buttons on the interface.
One button controls the passage of time on that timeline.
The other moves a pointer on the timeline to zero.



And the position of the counter on that timeline is shown as output.



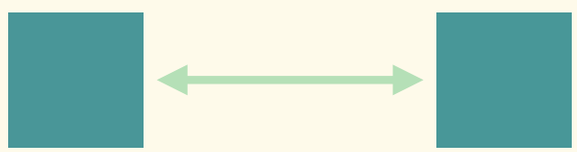
That’s a peek at what I see inside my mind when I think of a stopwatch.
There’s a little world behind this simple interface.

Interfaces

```
Stopwatch

- start
- stop
- reset

- watch_time
- counting?
```

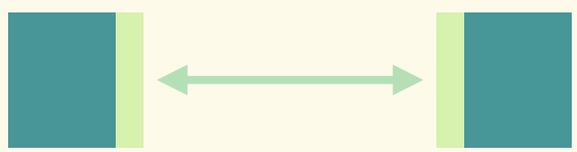
If you’ve written code, you might think about how to implement the stopwatch…
We could start with the commands (input) and questions (output) that the interface implies.
This is a just one way to look at the world functioning inside the stopwatch.

Interfaces



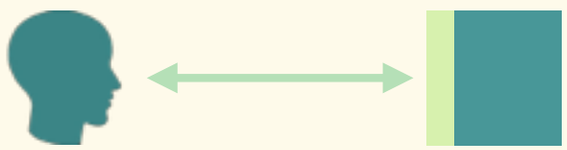
The stopwatch has an interface for people to interact with it, but interfaces can exist between any two things that interact.
(!) The word comes from “inter-”—like “interact”—and “face” like “surface”.
They are the surfaces that mediate interactions.

Interfaces



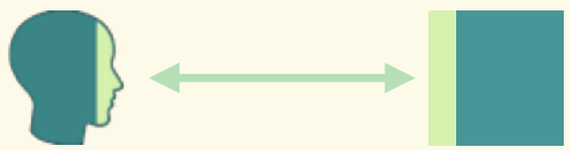
The stopwatch has an interface for people to interact with it, but interfaces can exist between any two things that interact.
(!) The word comes from “inter-”—like “interact”—and “face” like “surface”.
They are the surfaces that mediate interactions.

Interfaces



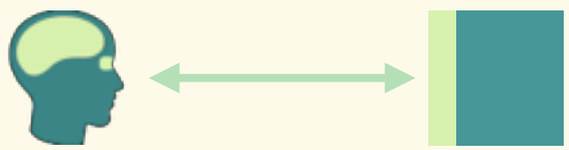
In the case of a human (or user) interface, it's (!) *your* face doing the interacting…
Or probably (!) more accurately your brain and your eyes… (!) and your hands.
Your human interfaces are *interacting* with the interface of the thing in question.

Interfaces



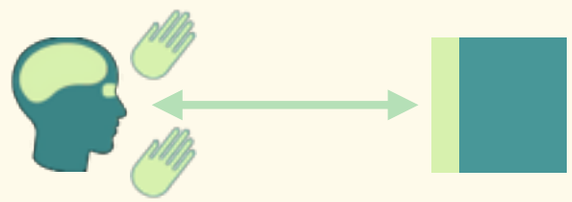
In the case of a human (or user) interface, it's (!) *your* face doing the interacting…
Or probably (!) more accurately your brain and your eyes… (!) and your hands.
Your human interfaces are *interacting* with the interface of the thing in question.

Interfaces



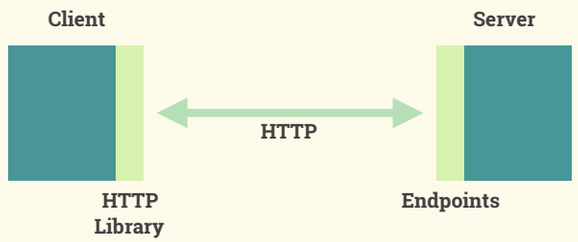
In the case of a human (or user) interface, it's (!) *your* face doing the interacting…
Or probably (!) more accurately your brain and your eyes… (!) and your hands.
Your human interfaces are *interacting* with the interface of the thing in question.

Interfaces



In the case of a human (or user) interface, it's (!) *your* face doing the interacting…
Or probably (!) more accurately your brain and your eyes… (!) and your hands.
Your human interfaces are *interacting* with the interface of the thing in question.

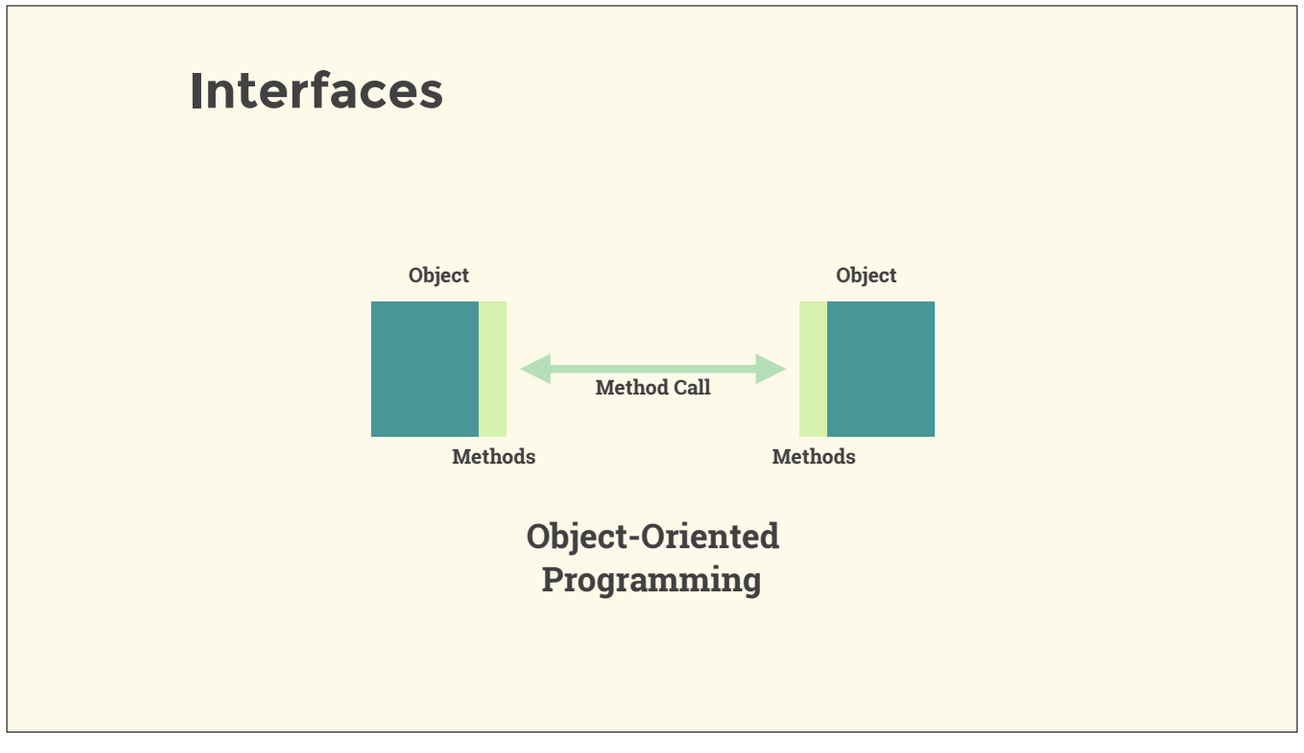
Interfaces



HTTP Interactions

What about some *non*-user interfaces?

(!) If you're familiar with HTTP interfaces, you can probably label all of these parts…



Or object-oriented programming, which is all about objects interacting with other objects via method calls.

Interfaces

```
java.util
public interface List<E>

- boolean  add(E e)
- void     clear()
- boolean  contains(Object o)
- E        get(int index)
- E        remove(int index)
- int      size()
```

And yes, code interfaces as well. If you're familiar, you can see that they describe the same types of things.
The "surface" exposed by this interface is its list of methods and their implied behavior.
Any object that puts on the face of a List will allow things to interact with it in *this* way.

Interfaces

Implementation

```
class Person

  def first_initial
    first_name.first.upcase
  end

  def last_initial
    last_name.first.upcase
  end

end
```

Interface

```
Person

- first_initial
- last_initial
```

Just a quick side note that this is also true of programming languages without capital-I Interfaces. Like Ruby.

This “Person” concept has an interface formed by two methods.

Every Person can be interacted with in the same way.

Levels of Abstraction

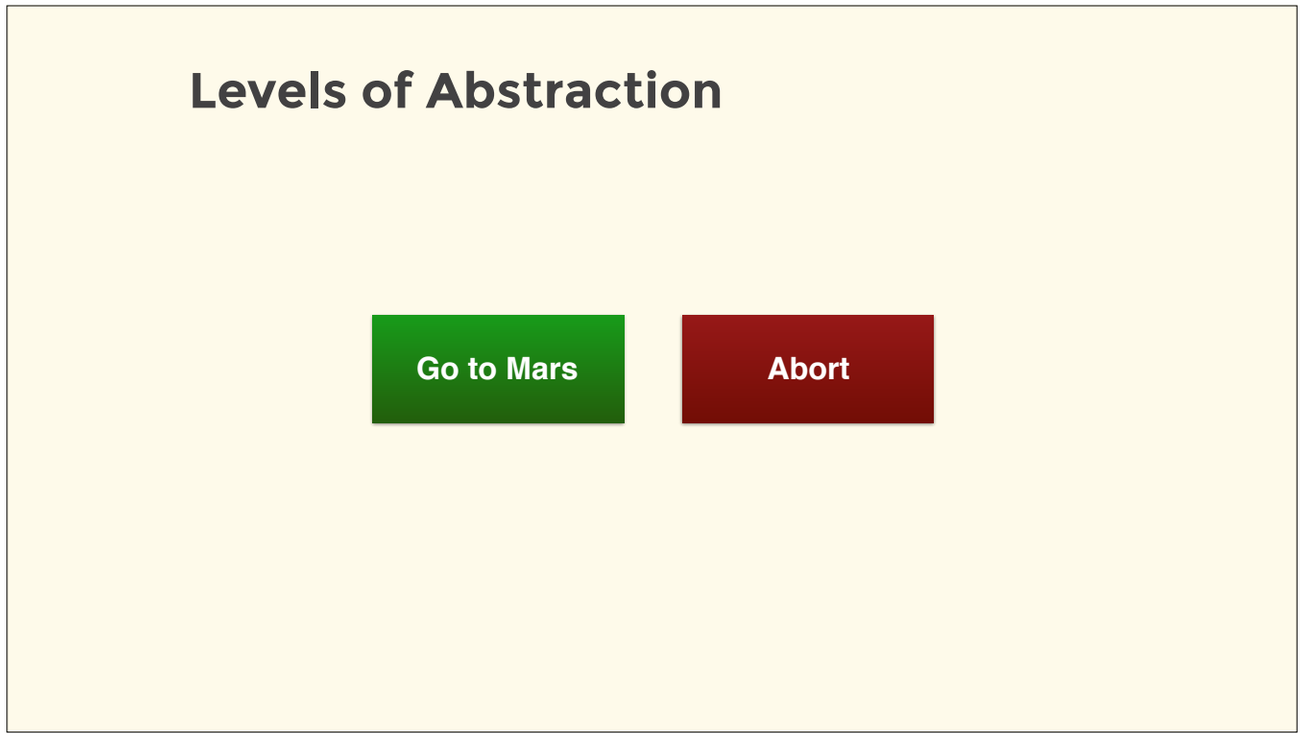
We’ll see more interfaces soon, but first let’s talk about this complicated term. (!)

…and yes, the name of this conference definitely inspired this talk. :)

Levels of Abstractions.io

😄

We’ll see more interfaces soon, but first let’s talk about this complicated term. (!)
…and yes, the name of this conference definitely inspired this talk. :)



What is a level of abstraction?

If we think about the worlds that live behind certain interfaces, like the timeline and details behind the stopwatch…

There will be those that are very simple on the outside, but probably very complex on the inside.

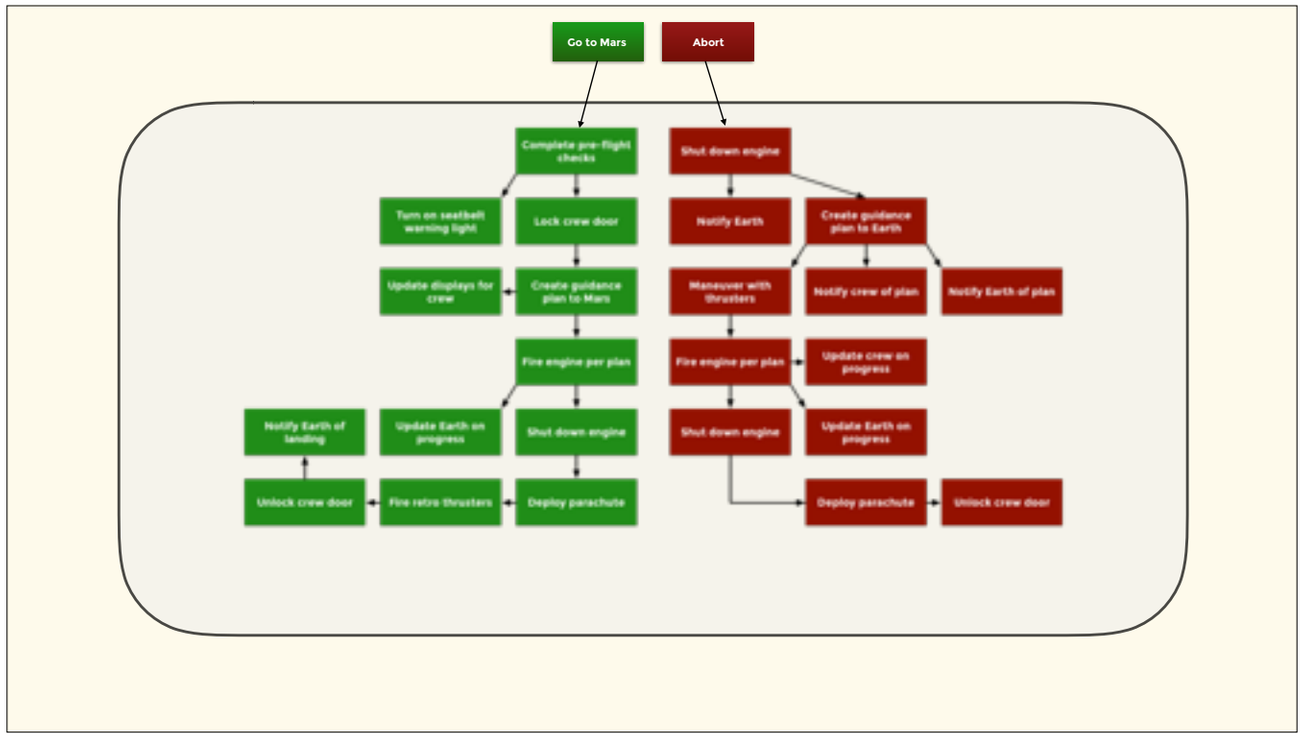
…and then those that are very complex on the outside.

One way to think about this difference is in terms of control.

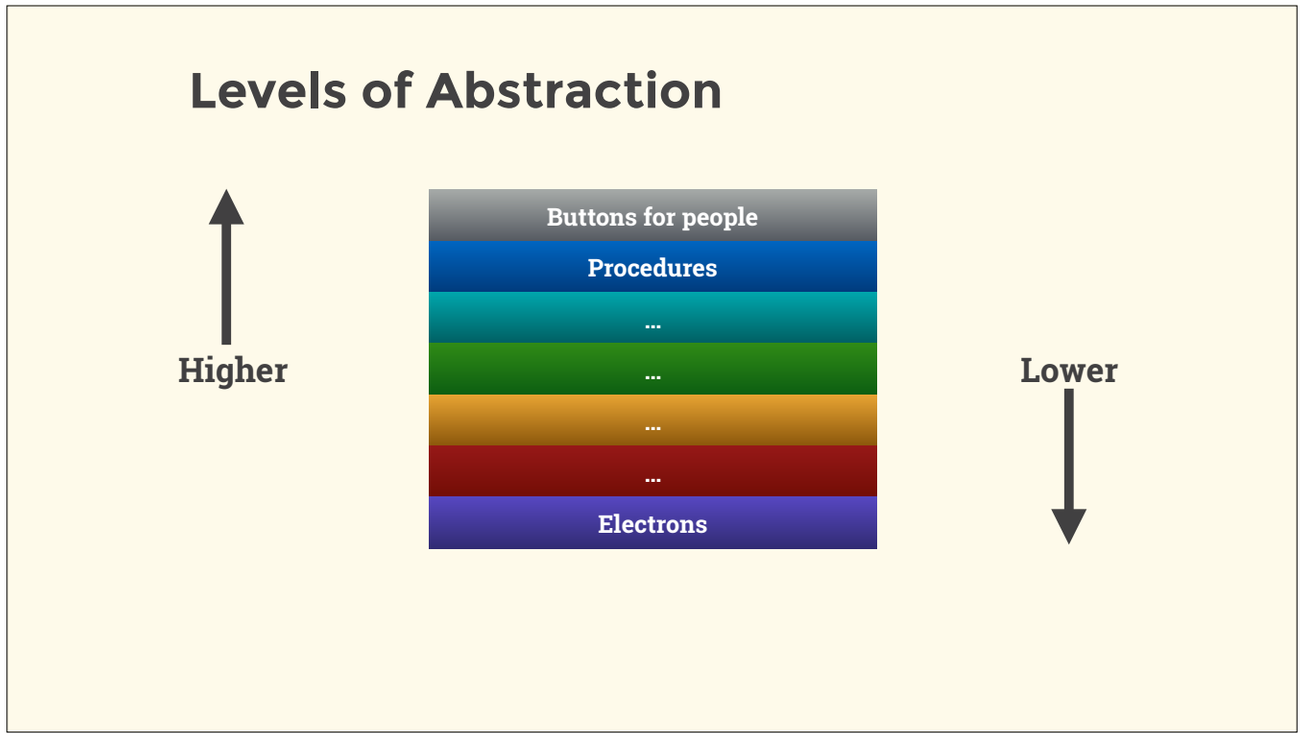
While this interface may offer a lot of control…

Levels of Abstraction

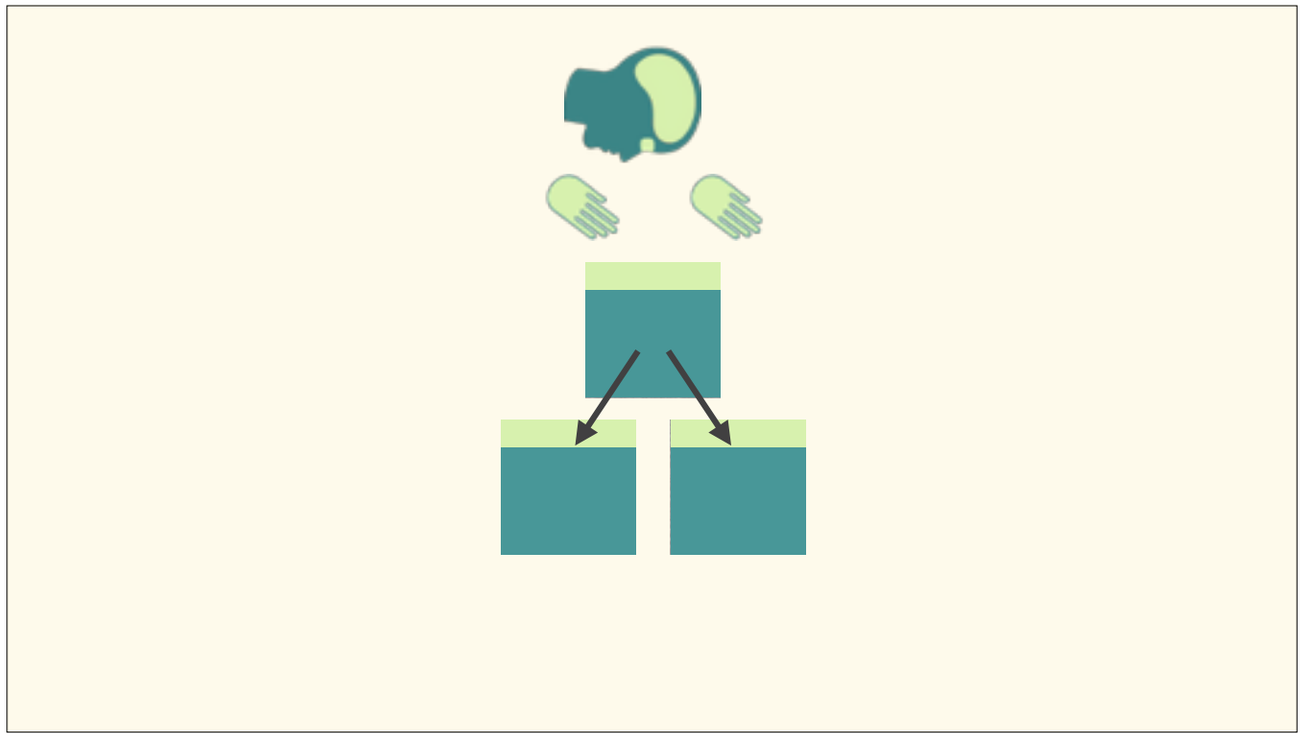
…this one offers very little control over what the system is doing.



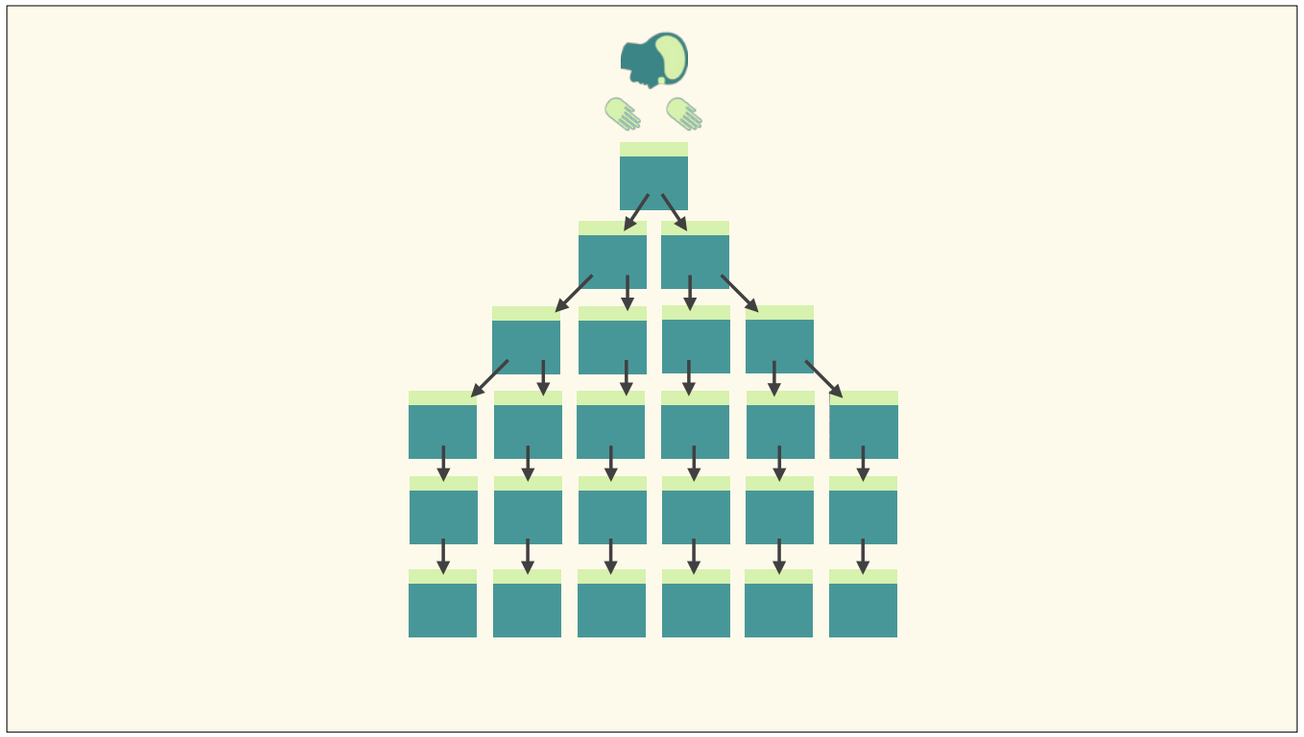
But if we peeked at the world one level below that simple interface, we'd see something more complex. Procedures…
But imagine that each of those steps were its own button.
Then a person would have more control over which ones happened, and in which order.



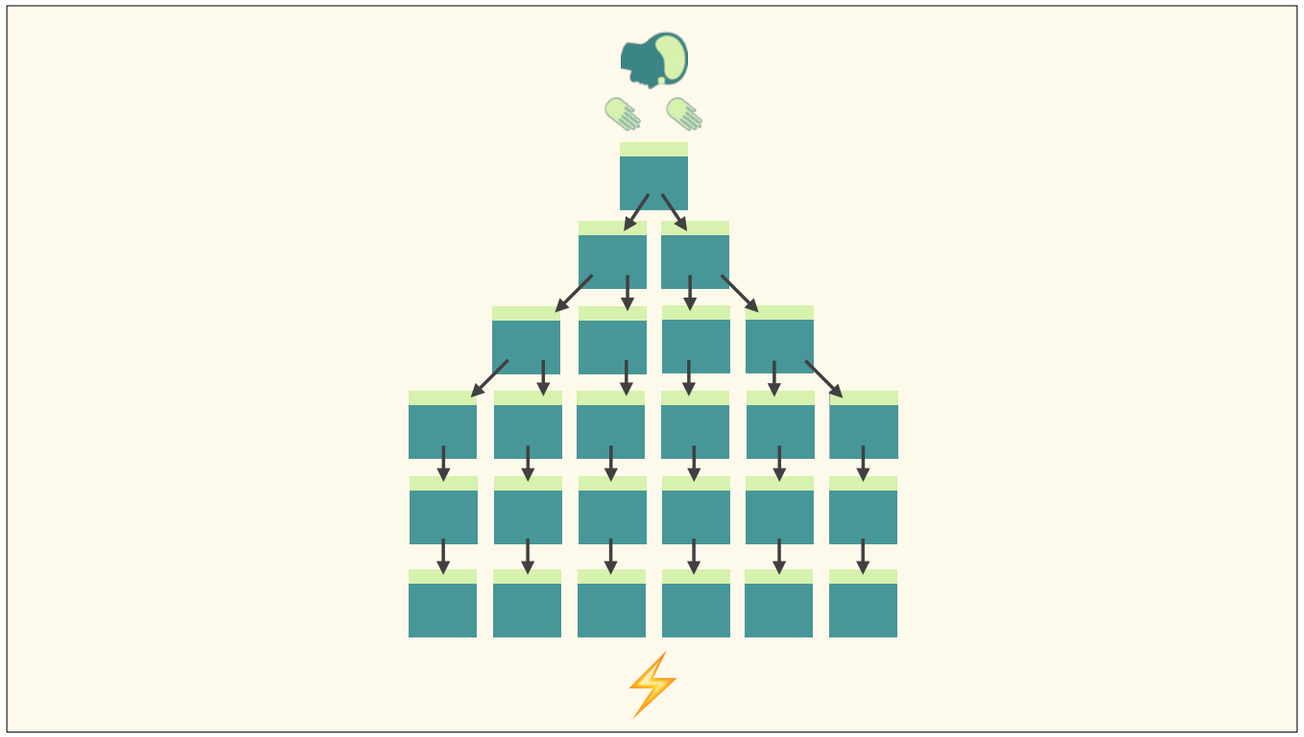
In this same way, we could dive deeper into any one of *those* buttons and find more parts of the system.
We call that a new level (or layer) of abstraction.
A lower level of abstraction means more control (and more complexity from the point of the view of the entire system).



I think of it a bit like this: if a user is interacting with a user interface at the highest level of abstraction…
Then that interface is implemented on top of a lower level of abstraction.

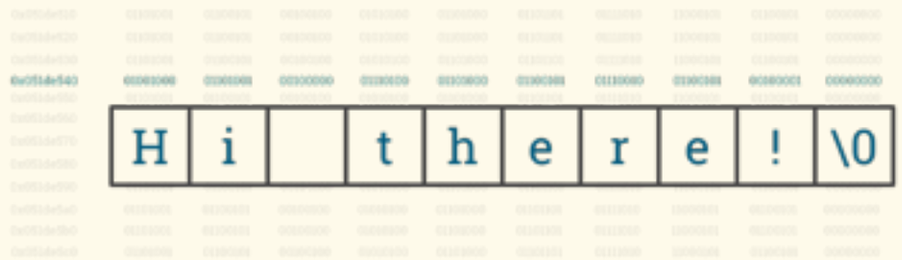


And *that* level is implemented on a bunch more levels, all the way down to the bits and volts.



And *that* level is implemented on a bunch more levels, all the way down to the bits and volts.

Levels of Abstraction




```
char *str;
str = malloc(32 * sizeof(char));
strcpy(str, "Hi there!");
```

Example… *memory management*.
When things need to be very fast…C or C++.
These languages ask the developer to be concerned with bits and bytes of memory—like, ones & zeroes represented by electron flows in a stick of silicon.
Worrying about how bytes of memory will be arranged is several levels of abstraction below, e.g. saving data from a web form…

Levels of Abstraction

```
current_user.update!(
  greeting: params['greeting']
)
```

Higher

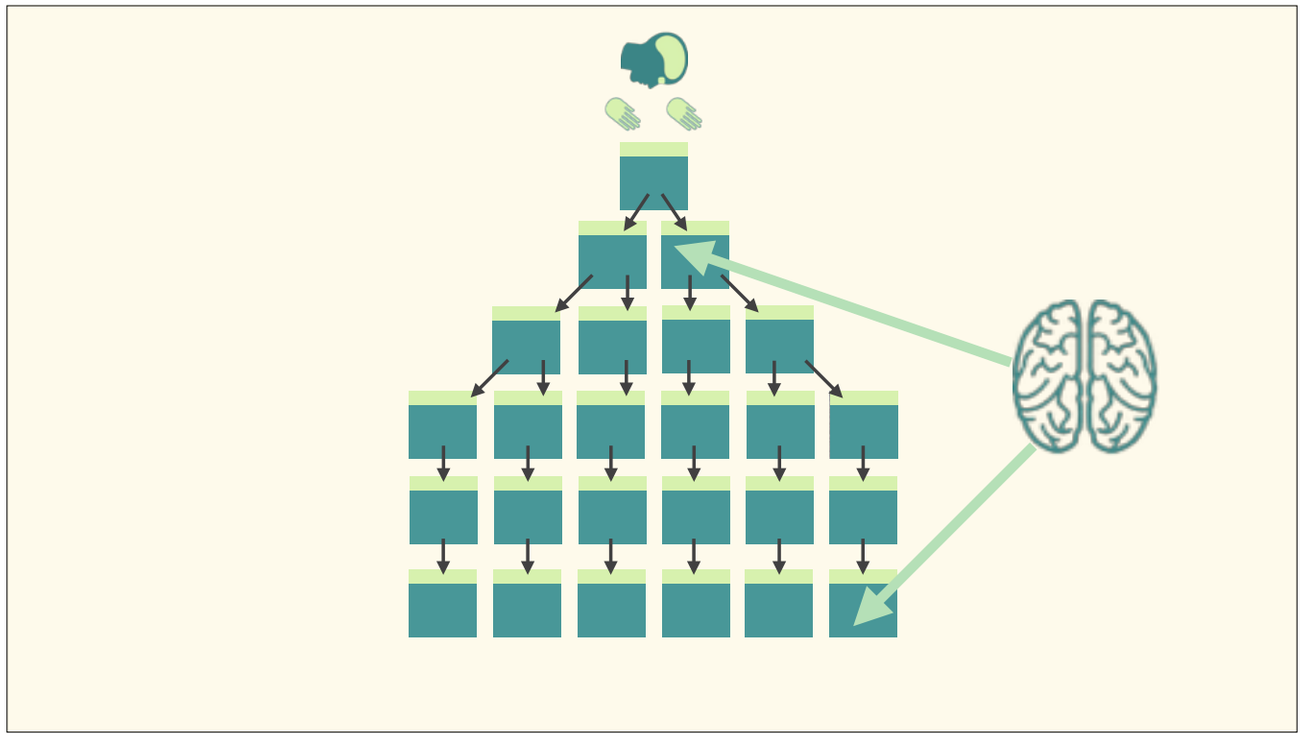
```
char *form;     /* 8 bytes */
long form_size; /* 8 bytes */
char mode;      /* 1 byte */
```

Lower

In one place a developer might be thinking about web form posts and user objects in a database.

In another, they’re thinking about which bits might line up with which bytes…

These are very different worlds. Or, rather, very different levels of the same world.



It’s a kind of mental gymnastics to reason about these two different levels *at the same time*. It’s understandable why two different levels, with different contexts, would be difficult to think about and change.

Building Worlds

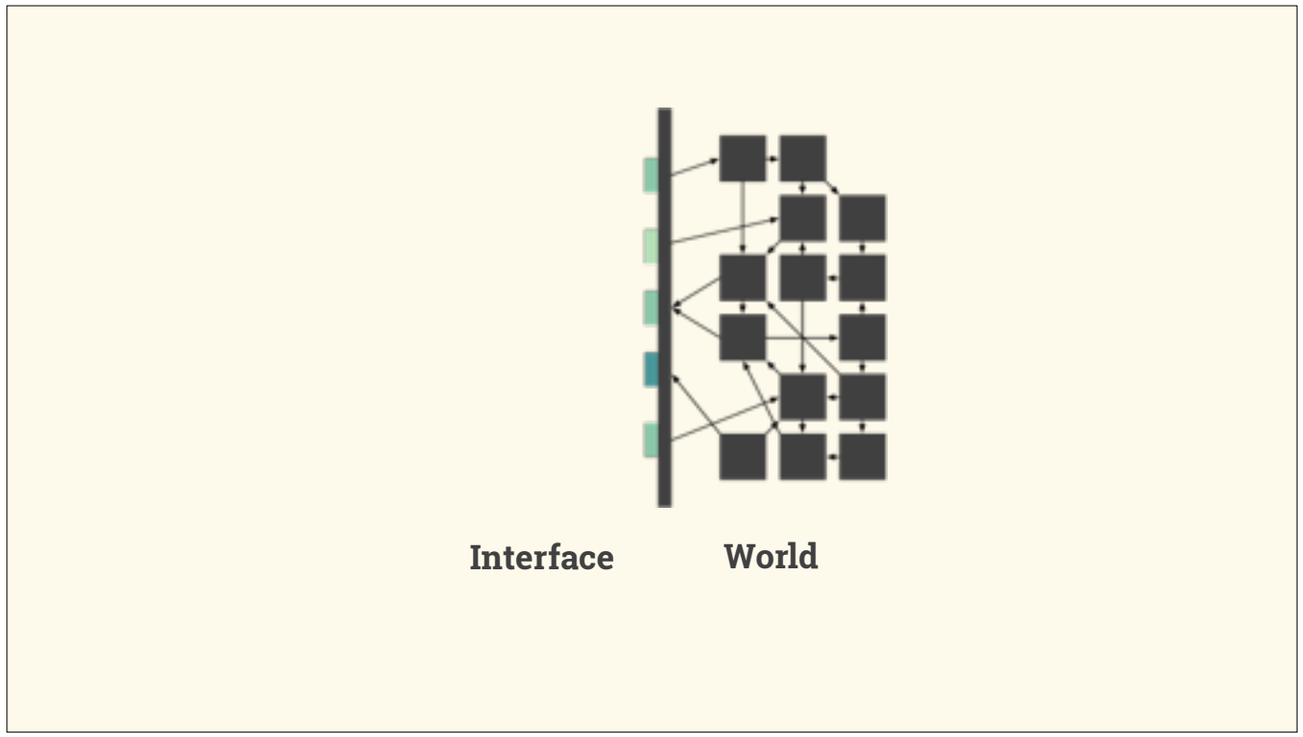
Now, with an idea of what interfaces and levels of abstraction are, let’s think about building worlds.

Building Worlds

For users, some interfaces are more difficult to learn than others, and may require more training or experience.

For instance, this watch's interface is more complex than our stopwatch. It has *several* inputs and outputs.

Think, for a second, about what the world must look like behind this interface … It has a lot more in it than the stopwatch's simple timeline, right?



Thought about another way, any *interface* to a system—the edge of the world—is necessarily simpler than the system itself.
The world that's behind the buttons and displays (or the function calls) is always more complex.

Interfaces

Implementation

```
class Person

  def first_initial
    first_name.first.upcase
  end

  def last_initial
    last_name.first.upcase
  end

end
```

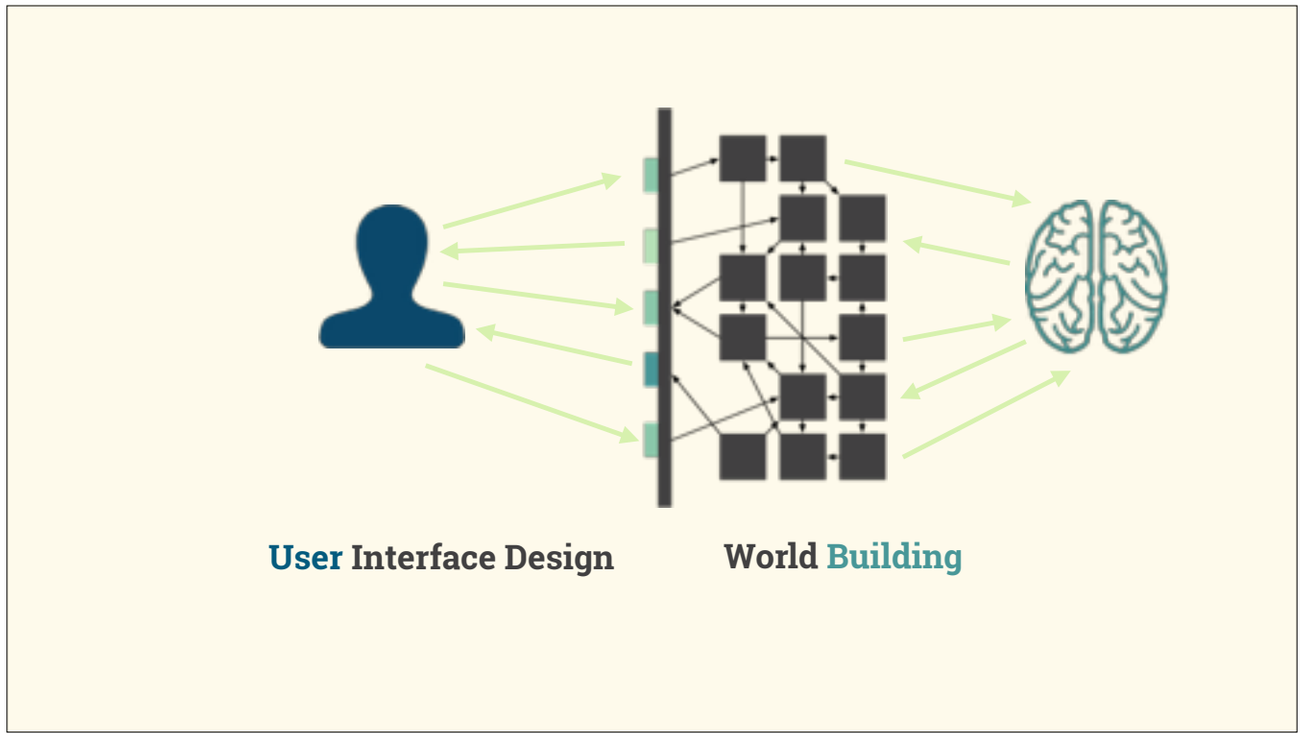
Interface

```
Person

- first_initial
- last_initial
```

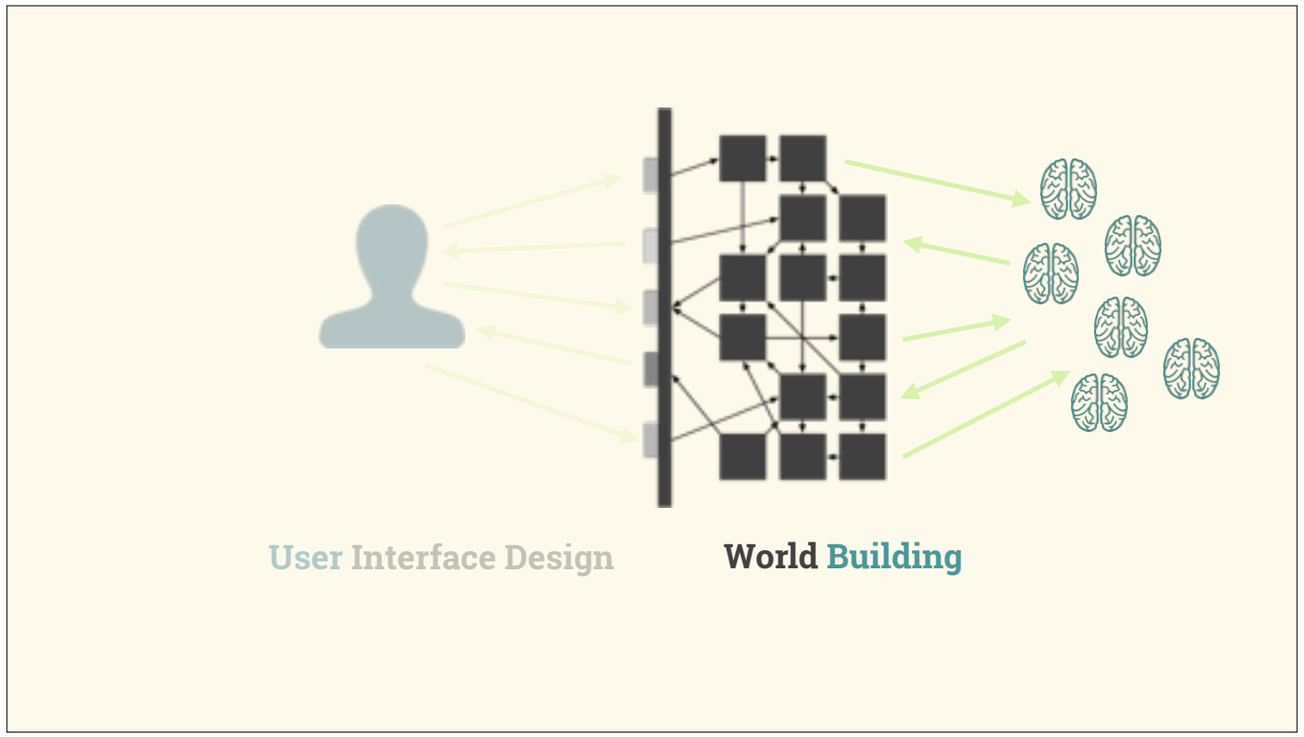
We saw this before, right?

Every module, class, or object in a system has an interface.

And every interface is simpler than the world beneath it.



Now, the art & science of creating good user interfaces…UI Design.
That discipline is very mature and… like empathy, usability, use cases and measuring task success.
But the details of building worlds—of creating modules, classes & objects—aren't often thought of in the same way.
One reason for this might be *this* idea here [point right]: that *one person* creates or changes the world at a time…

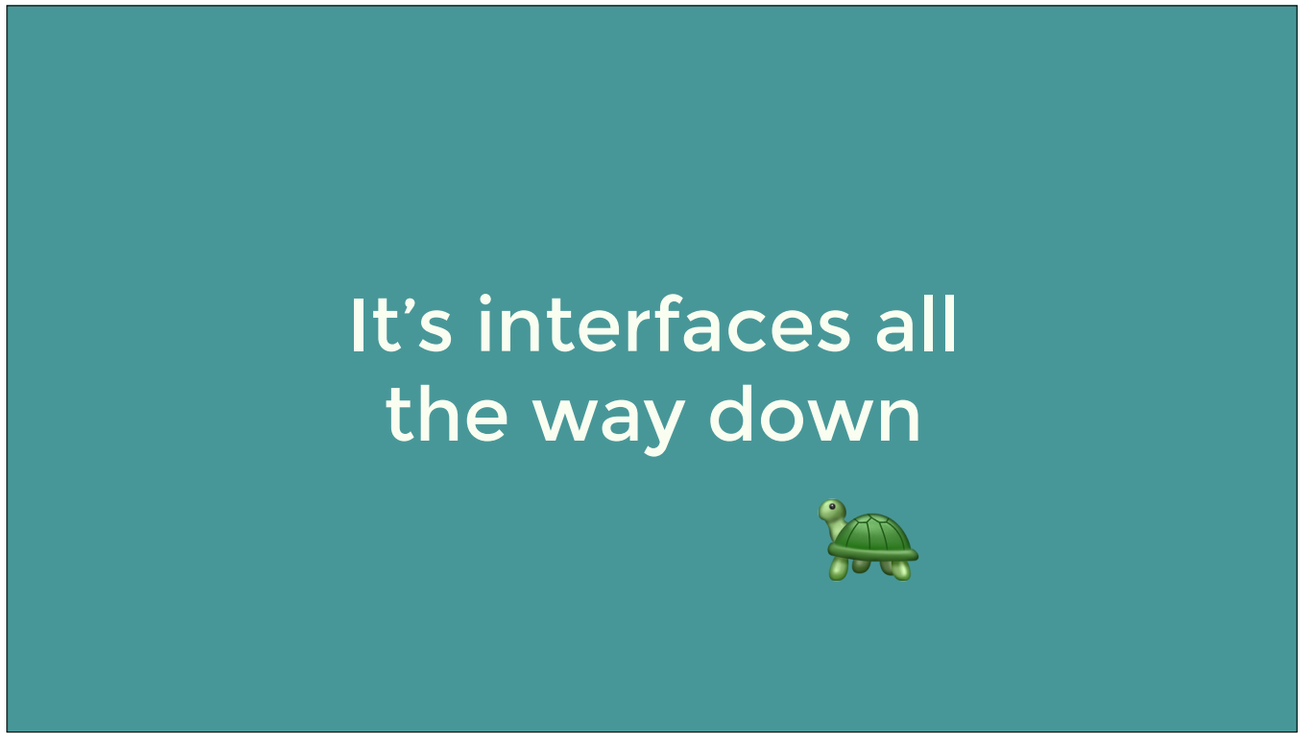


But the reality is that many minds contribute to the success of a system from behind the scenes.
…whether that’s a team of different people, or the same person tending the world over time…
How could we help these different minds effectively build a world together?

Building Worlds

1. It's interfaces all the way down
2. A good interface works at one level of abstraction
3. Coupling is expensive forever
4. Systems are for humans first, machines second

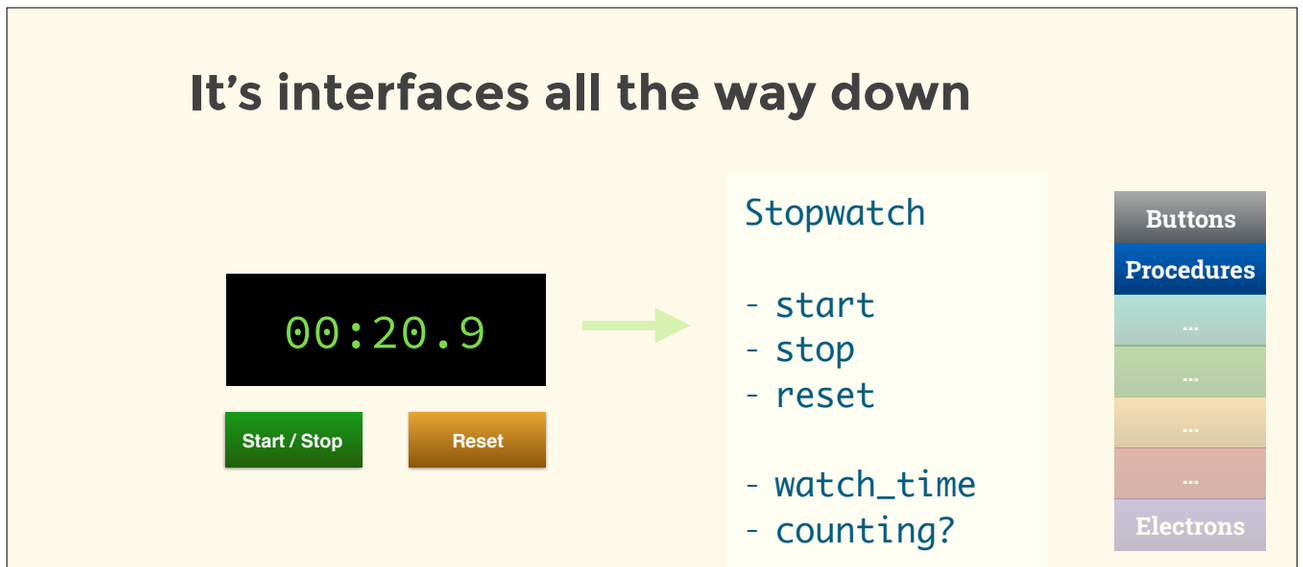
I think we may have some success if we start with these four propositions.
[read 4]
Let's spend some time with each of these, because these are really at the core of building better worlds.



First, it’s interfaces all the way down.

Recognizing that there are interfaces at every level of abstraction in a system…

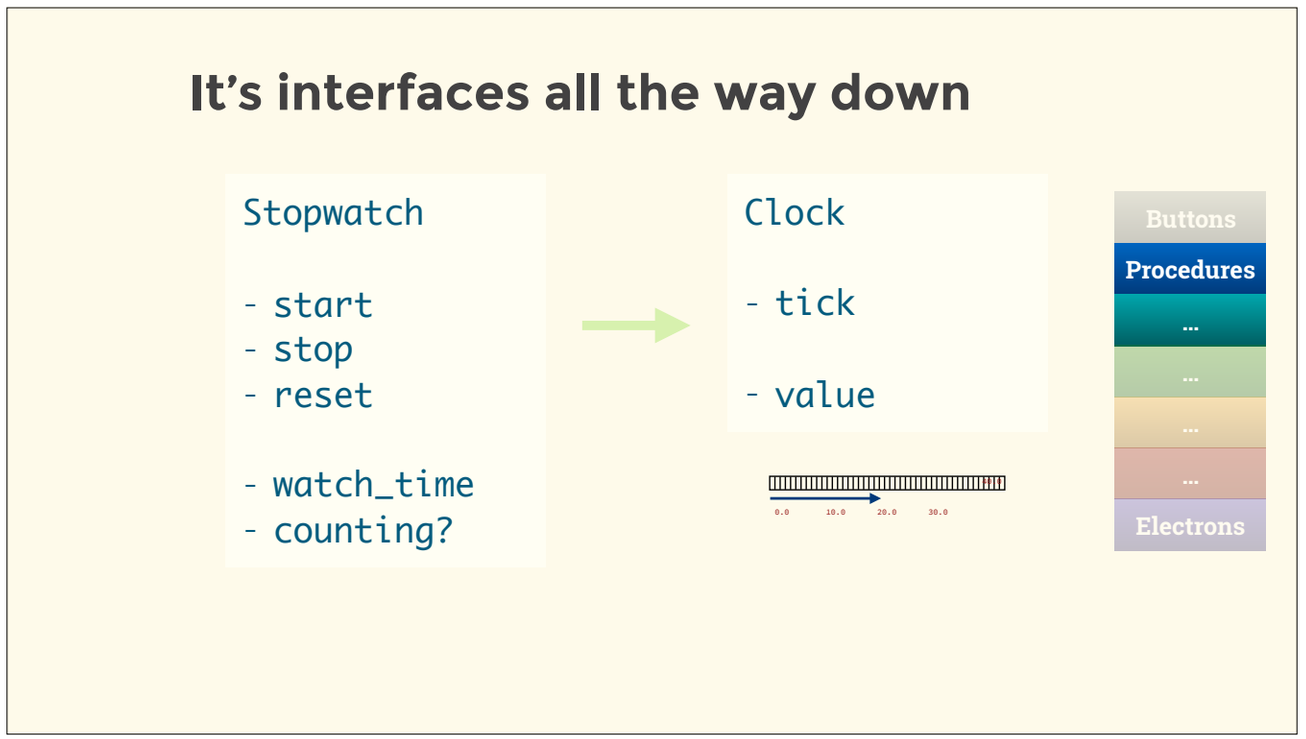
…means we have ample opportunities to make those interfaces better.



The stopwatch that has buttons and a display is the highest level of abstraction; and it’s an interface.

The next level down, though—the procedures or functions behind the buttons—also form an interface.

The user of the stopwatch may only ever see the first level, but the *developers* of the interfaces will see and change both levels.



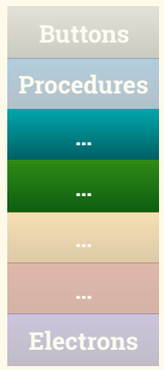
Similarly, if we dive one level deeper, behind the Stopwatch *functional* interface…
…we’d expect to find some version of the little world—the ever-increasing timeline.
So there might be a “clock” that just knows how to “tick” and return its current value. This clock is also an interface.

It’s interfaces all the way down

```
Clock

- tick

- value
```



```
# CLOCK_GETRES(2)

int clock_getres(
 clockid_t clk_id,
 struct timespec *res);

int clock_gettime(
 clockid_t clk_id,
 struct timespec *tp);

int clock_settime(
 clockid_t clk_id,
 const struct timespec *tp);
```



One level deeper yet, there’s another interface, modeling clock & counting behavior on the CPU.

CPUs just process instructions at a specific clock speed, which…

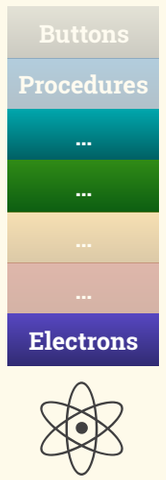
…is another way of saying that they can measure and control the (!) [grin] flow of electrons in a computer.

It’s interfaces all the way down

```
Clock

- tick

- value
```



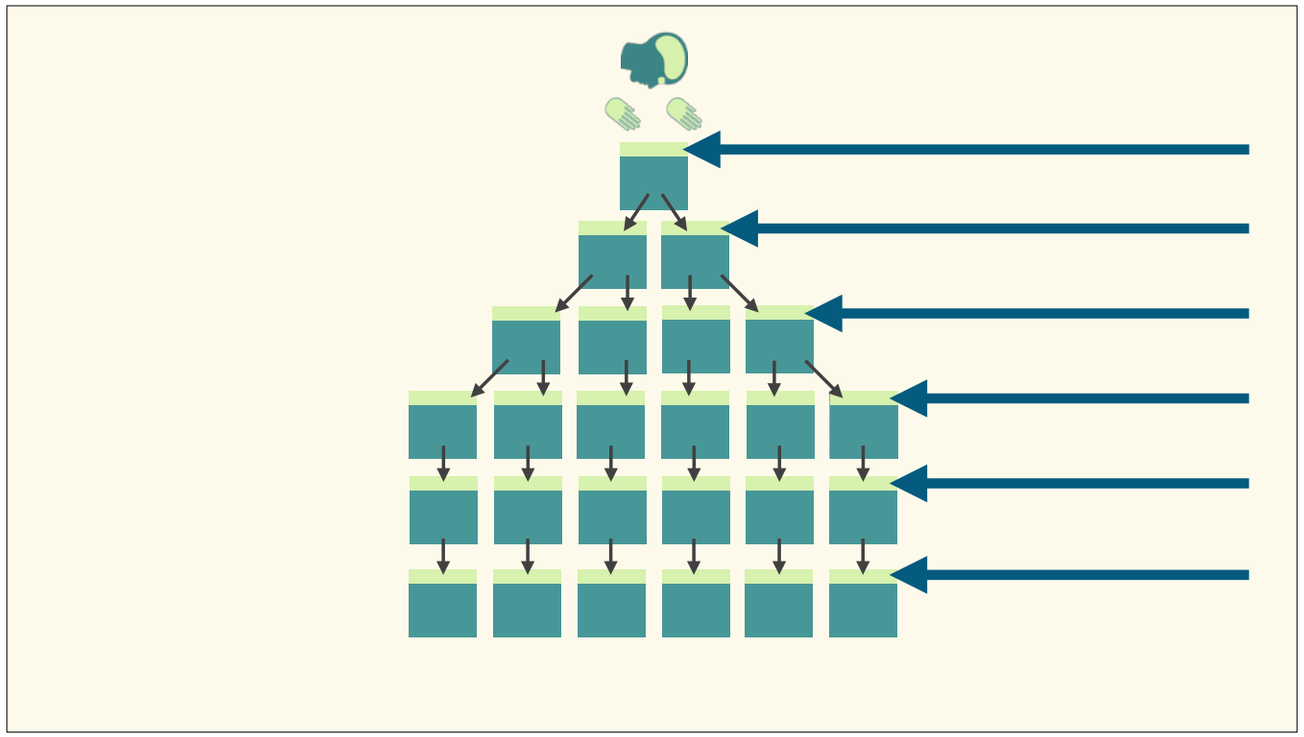
```
# CLOCK_GETRES(2)

int clock_getres(
 clockid_t clk_id,
 struct timespec *res);

int clock_gettime(
 clockid_t clk_id,
 struct timespec *tp);

int clock_settime(
 clockid_t clk_id,
 const struct timespec *tp);
```



One level deeper yet, there’s another interface, modeling clock & counting behavior on the CPU.

CPUs just process instructions at a specific clock speed, which…

…is another way of saying that they can measure and control the (!) [grin] flow of electrons in a computer.



We knew this from before: each level of abstraction can have its own interfaces, whether those are
for a user to push a button or a program to read some bytes from memory. They’re all interfaces.

It’s interfaces all the way down

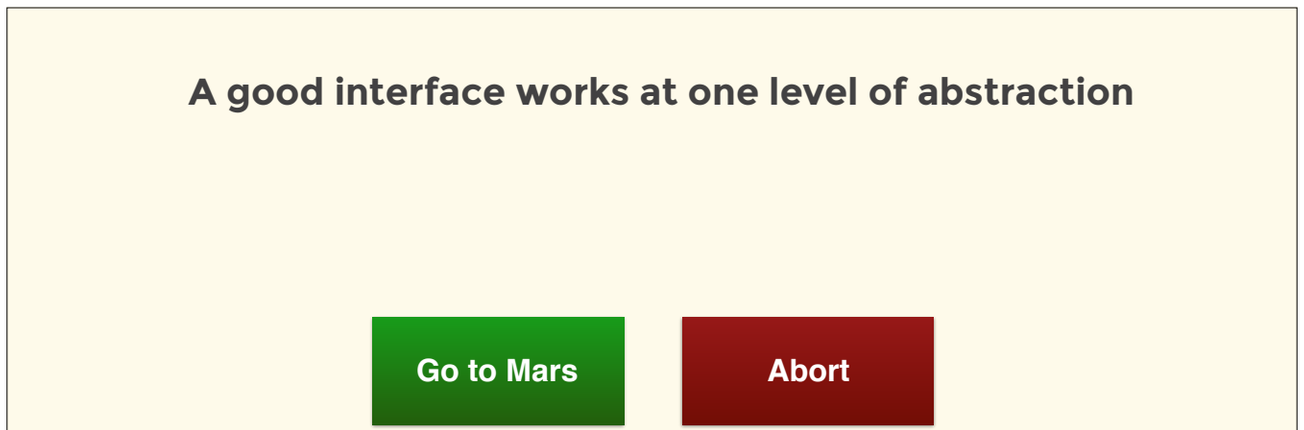
- There are interfaces at many levels of abstraction
- Each of these interfaces is a chance to be intentional
- These are opportunities!

Okay, so there are different interfaces covering many levels of abstraction…
Just as with the buttons and displays, each of these lower-level interfaces is a chance to be intentional…
They are *opportunities* to create better worlds by creating good interfaces.

A good interface works at one level of abstraction

Next, a good interface works at one level of abstraction.
An interface is not bound to a single level of abstraction—it can span multiple.
But when it sticks with just one, it is better and easier to use.

A good interface works at one level of abstraction



If we go back to our Martian spacecraft interface…
It’s fairly clear that these two buttons are operating at the same level of abstraction.
Both have roughly the same control over the underlying system (not much), and trigger similar sets of procedures.

A good interface works at one level of abstraction

But what if we replaced one button with the procedures it uses at a lower level of abstraction?
…then the interface would be *awkward* and probably more error-prone, if the crew had not been trained on it.
It would be more difficult to *think about* and *control* what's happening to the spacecraft with these two levels of abstraction next to one another.

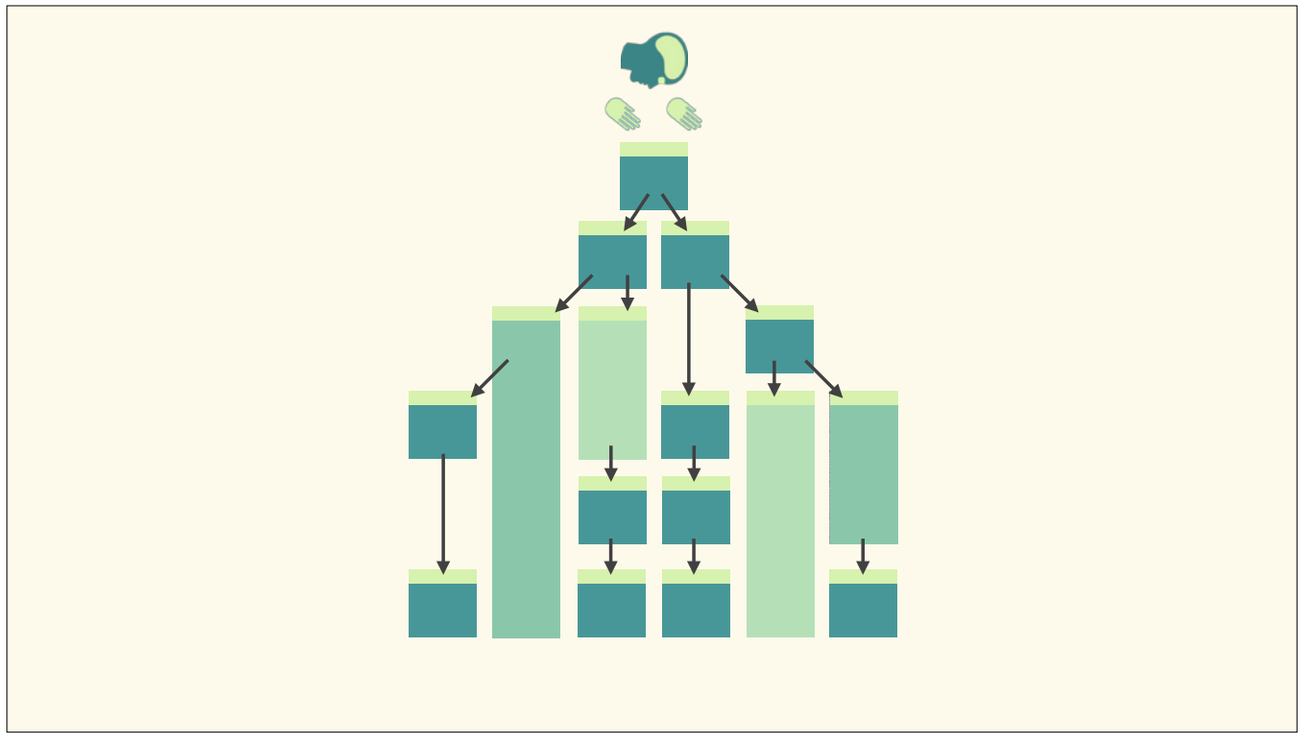
A good interface works at one level of abstraction

But if we went further and replaced both buttons with the interfaces from the lower level of abstraction…

…*and* we trained the crew with instructions & step-by-step manuals…

This might be a reasonable interface for the spacecraft (this *is* closer to what we've seen in e.g. "Apollo 13").

Depending on the level of control we want the users to have, we can make a decision about the level of abstraction for the interface.



We’d get into this situation where interfaces are trying to offer control of many levels of abstraction,
or skipping many levels and reaching way too far down into bits & bytes when they should be thinking about people. Or databases.

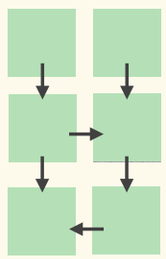
A good interface works at one level of abstraction

- **Interfaces are better within one level of abstraction**
- **Complexity means more training & experience needed**
- **We can use these insights to improve our interfaces!**

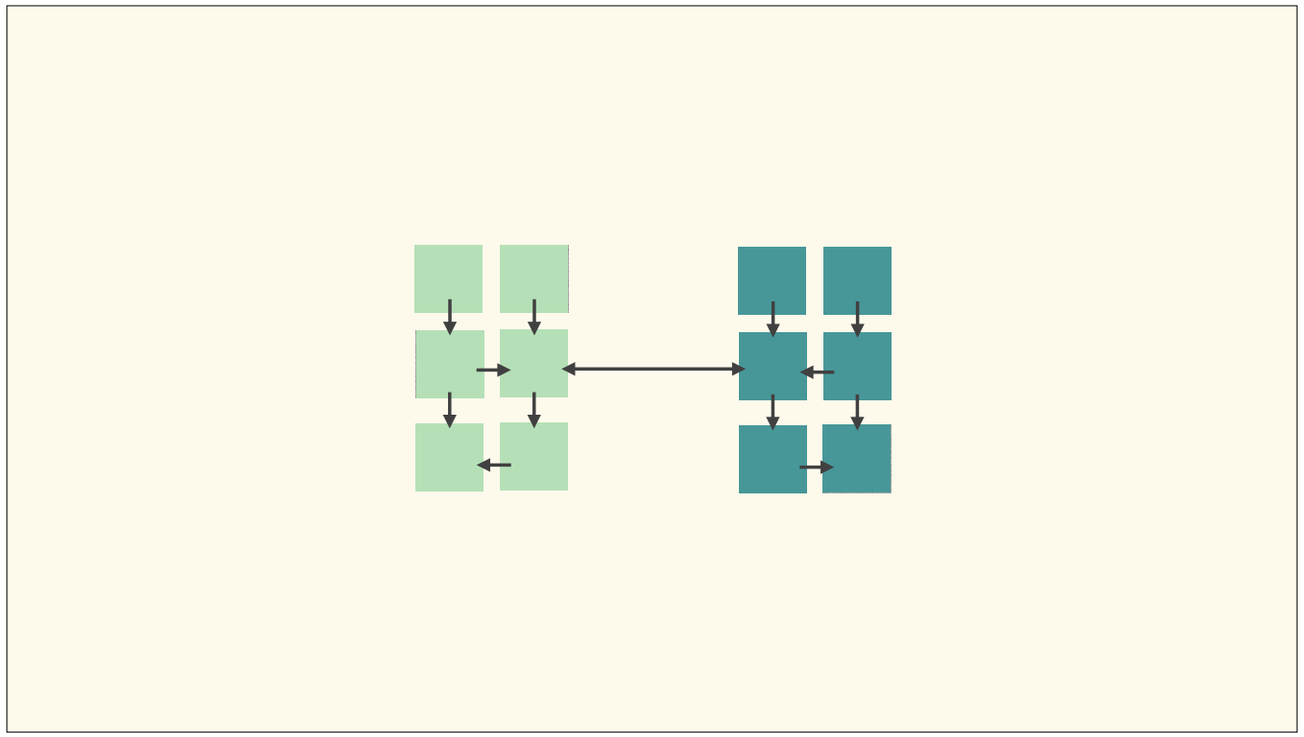
So we'll want to keep each of our interfaces to one level of abstraction when possible.
And we should also be aware of the tradeoff between lower-level interfaces' complexity and the time & training required to use them well.
I think we could use these insights to improve!

Coupling is expensive forever

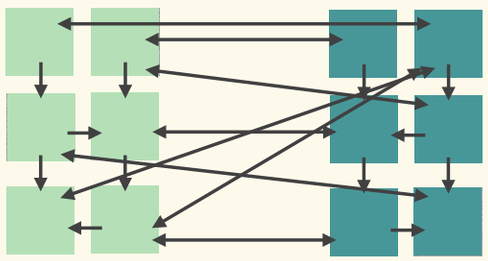
Coupling is what happens *between* interfaces and components. The arrows between the boxes.



When we have a small system that only does a few things, coupling makes sense and comes with the territory.
The display of a stopwatch is coupled to its internal counting mechanism. Because that’s the whole point.
But what if you wanted to build an alarm clock, to sit by your bed, and you think: “I know! I can reuse the internal clock from the stopwatch!”

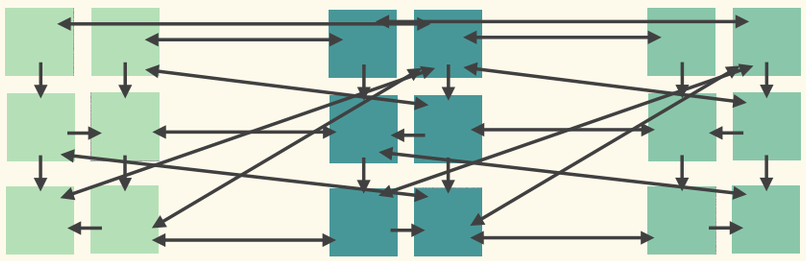


Now…two systems, …one depends on…other.
When one part…depends on another (via a method call, data sharing, sequencing or anything), we say that they are “coupled”.
Anytime something changes on one side of the coupling, things may also need to change on the other side.
This means that small changes can turn into larger necessary changes *simply because components are coupled*.



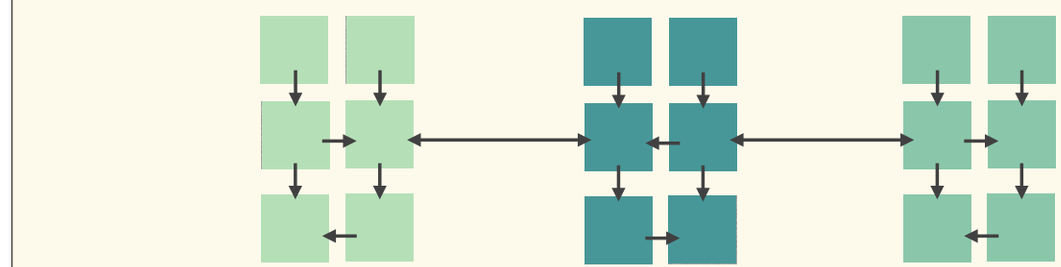
Tight Coupling

Inevitably, as systems grow…coupling continues…[toward messiness].
When many parts rely on many other parts in a system, we think of the coupling as “tight”.
It will take more effort to disentangle them, and every change in one can trigger a cascade of potential changes in the other.
This makes it more expensive to make small changes to one side, because it ends up triggering small changes everywhere at once.



Tight Coupling

And if you’re building systems in a business, like many of us are, time pressure means that *more* of this tight coupling happens.
New things will be added, relying on older ones…coupling will increase.
I feel like a lot of presenters, at this point, would explain the *one true solution* to a problem like this.
But there really isn’t one. The true solution is to do a lot of hard work to pull these things apart…



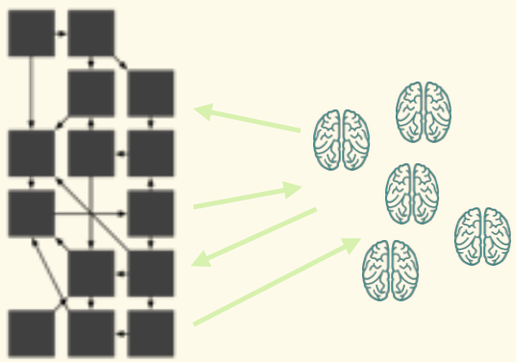
Loose Coupling

…and achieve loose coupling. Where each…relies…only very specific ways that are easy to understand.
And easy to change. …cheaper. Small changes can stay small.
But…will increase again. …real, sustained human effort…stay loose
And it won’t always make sense to pay that cost when making a change. It’s an endless loop and you pay either way.
That’s why I say: coupling is expensive forever.

Systems are for humans
first, machines second

We understand some things about interfaces and systems now. But even if we learn all of those things really well,
we might still make interfaces that are less than great.
This, here, is the single most important principle to take away today. And every day. Bring it to every place where worlds are built.
Systems are for humans first, machines second. Systems are for humans first, machines second.

Systems are for humans first, machines second



All worlds beyond a certain complexity will need to change.
Many different minds will need to change them; again: whether different people or the same person weeks or months apart.
So the ease of changing a world like this depends directly on how difficult it is to understand *for other minds*.
Humans must be our first audience for the interfaces & parts of this world if we want it to change smoothly.

Systems are for humans first, machines second

- **Attention span**
- **Intensity of focus**
- **Discontinuous attention**
- **Different desires**
- **Mistakes**
- **Shenanigans**

Humans…UI designers know…are a weird design constraint…
We have limitations, like [read]—and that's just if everything goes right—[shenanigans]…
If you were designing only for machines, you really wouldn't have to worry about any of these things.
But that's how we end up with brittle, confusing pieces in our worlds when we're trying to change them.

Systems are for humans first, machines second

- **Employ empathy**
- **Strive for simplicity**
- **Think about usability**
- **Plan use cases**
- **Measure task success**
- **Get feedback!**

So how can we create interfaces & build worlds for humans first?
The good news is that someone—or lots of someones—have already been here, figuring out how to create good interfaces for humans.
We can start with a lot of the things UI designers already do…[read]

Systems are for humans first, machines second

- **Hard work**

The other good news is that the only thing between you and creating really great systems for humans first…
…is a lot of hard work.
This means it’s an opportunity! …for sustained creative effort. …to find better ways to build things. …to find ways to be a better human.

Feeling Great

I’ve started using subjective words like “awkward”, “fragile”, “human”…
So now’s a good time to admit that part of our success building little worlds with other minds will depend on other builders’ *feelings*.
Feelings of an almost human nature…

Feeling Great

When you’re trying to solve a problem, when do you feel good?
Is it when you’re confused and stuck? Or when things are (!) easy to understand?
When the tiniest change breaks something in an unexpected way? (!) Or when simple changes do what you expect?
When things go well, (!) don’t you feel more confident? (!) Happy? Empowered? *Feelings!*

Feeling Great

- **Easy to understand**

When you’re trying to solve a problem, when do you feel good?
Is it when you’re confused and stuck? Or when things are (!) easy to understand?
When the tiniest change breaks something in an unexpected way? (!) Or when simple changes do what you expect?
When things go well, (!) don’t you feel more confident? (!) Happy? Empowered? *Feelings!*

Feeling Great

- **Easy to understand**
- **Simple changes have obvious results**

When you’re trying to solve a problem, when do you feel good?
Is it when you’re confused and stuck? Or when things are (!) easy to understand?
When the tiniest change breaks something in an unexpected way? (!) Or when simple changes do what you expect?
When things go well, (!) don’t you feel more confident? (!) Happy? Empowered? *Feelings!*

Feeling Great

- **Easy to understand**
- **Simple changes have obvious results**
- **Builds confidence in tending the world**

When you’re trying to solve a problem, when do you feel good?
Is it when you’re confused and stuck? Or when things are (!) easy to understand?
When the tiniest change breaks something in an unexpected way? (!) Or when simple changes do what you expect?
When things go well, (!) don’t you feel more confident? (!) Happy? Empowered? *Feelings!*

Feeling Great

- **Easy to understand**
- **Simple changes have obvious results**
- **Builds confidence in tending the world**
- **Inspires happiness & empowerment**

When you’re trying to solve a problem, when do you feel good?
Is it when you’re confused and stuck? Or when things are (!) easy to understand?
When the tiniest change breaks something in an unexpected way? (!) Or when simple changes do what you expect?
When things go well, (!) don’t you feel more confident? (!) Happy? Empowered? *Feelings!*

Easy to understand

I love feeling great. Especially when I’m just trying to solve *one* problem and every piece just *fits* with my brain.

Do you know that feeling?

When we’re working with multi-level fractal worlds with innumerable interfaces and behaviors, if you feel *that—the just-fits-ness*…

…you can bet that someone put some *effort* into making things easy for you.

Easy to understand

What makes something easy to understand?
How about if we think of…as reading a fantasy novel? You write the novel and I'll read it. (!)
First, I want to know who the people are. (!)
Then I want to know how things work—Is there magic? How strong is gravity? Are some people immortal? That's interesting. (!)
Then…the stage: What's going on? Who's interacting…?
All of these things help me understand what's going on…

Easy to understand

- **Who are these people? What do they do?**

What makes something easy to understand?
How about if we think of…as reading a fantasy novel? You write the novel and I'll read it. (!)
First, I want to know who the people are. (!)
Then I want to know how things work—Is there magic? How strong is gravity? Are some people immortal? That's interesting. (!)
Then…the stage: What's going on? Who's interacting…?
All of these things help me understand what's going on…

Easy to understand

- **Who are these people? What do they do?**
- **How do things work in this world?**

What makes something easy to understand?
How about if we think of…as reading a fantasy novel? You write the novel and I'll read it. (!)
First, I want to know who the people are. (!)
Then I want to know how things work—Is there magic? How strong is gravity? Are some people immortal? That's interesting. (!)
Then…the stage: What's going on? Who's interacting…?
All of these things help me understand what's going on…

Easy to understand

- **Who are these people? What do they do?**
- **How do things work in this world?**
- **What's going on right now in the story?**

What makes something easy to understand?
How about if we think of…as reading a fantasy novel? You write the novel and I'll read it. (!)
First, I want to know who the people are. (!)
Then I want to know how things work—Is there magic? How strong is gravity? Are some people immortal? That's interesting. (!)
Then…the stage: What's going on? Who's interacting…?
All of these things help me understand what's going on…

Easy to understand

- **Task: Add a new chapter to the middle of the story**
  (Hint: Modify a part of the working system)
  - **Read the intro guide**
    (Hint: Read the documentation)
  - **Understand the scene**
  - **Try changing something**
  - **Repeat until it works**

But then I want to change something in your book—I want to add a chapter. Also: every reader will skip to any chapter they want, arbitrarily.
…you can’t rely on a narrative intro. But you could supply an intro guide. Then I could read it.
After I understand the basics, I’ll probably just try to change a few things and see what happens.
Then I’ll do that again until the story makes sense.

Easy to understand

- **Good documentation**
  - **Concise**
  - **Common use cases considered**
  - **Good examples with links to further reading**
  - **Clearly defined & explained jargon**

What makes a good intro guide? Well, ideally it’s short, because I really just want to play around with the story.
It should also be relevant to me, a common chapter-adder hacking up your story. Consider my use case!
…great if it didn’t say things like “sometimes magic happens”…but instead described *how* magic can happen…
Lastly, …great if it gently introduced me to terms…. [clever: slines/mobes/extramuros]

Easy to understand

- **Simple mechanics**
- **Allows focus on one thing at a time**
- **Doesn't hurt my brain**
- **Fits with its domain**

Some other things that can make it easier to understand: a *simple* narrative… (!)

> “But some stories, small, simple ones about setting out on adventures or people doing wonders, tales of miracles and monsters, have outlasted all the people who told them…”
>
> Neil Gaiman

The small, simple stories seem to outlast them all…

Easy to understand

- Simple mechanics
- Allows focus on one thing at a time
- Doesn't hurt my brain
- Fits with its domain

I really want a story that lets me focus, take it all in, experience what happens, and move on.
I don't want to hold every detail of the universe in my head at once. (!)

Easy to understand

```
func Pipe() (*PipeReader, *PipeWriter)

client := &http.Client{
  CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
```

Nor do I really want to keep track of the fact that some magic symbols in the text mean that some sub-world is being invoked…
…and I have to keep track of what's going on there in addition to the main storyline. Ugh.

Easy to understand

- **Simple mechanics**
- **Allows focus on one thing at a time**
- **Doesn't hurt my brain**
- **Fits with its domain**

Lastly, the people probably shouldn't just start speaking a new language halfway through the book. Or change their core personalities.
If the book is in a series and I understand the canon, I should feel at home. I should be able to extend it.
It should use and build on familiar concepts rather than inventing a new way to fly in Chapter 27 for no apparent reason.

Naming

…apparently I have feelings about how stories should be written.

Anyway, what were we talking about? Computers?

I want to riff a bit on one of the topics closest to my heart … … naming.

Naming

```
FileUtils.mkdir_p(dir, opts)
FileUtils.rmdir(dir, opts)
FileUtils.ln_s(old, new, opts)
FileUtils.cp_r(src, dest, opts)
FileUtils.rm_rf(list, opts)
FileUtils.chmod_R(mode, list, opts)
FileUtils.chown_R(user, group, list, opts)
```

Fitness for domain

I mentioned this nebulous idea that your story (or system) shouldn’t hurt my brain. But who am I? And what causes brain hurt?
I think *names* can hurt brains. …& feelings. Not-great names, specif.
One aspect of naming is fitness for domain: do the names in your interfaces make sense within their context?
Ruby has this module called FileUtils where… If you didn’t know… Was that the best decision…?

Naming

- **Nouns**
  - **Directories**
  - **Links**
  - **Files**
  - **File Attributes**
- **Verbs**
  - **Create**
  - **Copy**
  - **Move**
  - **Delete**

Fitness for domain

Would it be better to approach the interface names afresh, breaking them into nouns & verbs for the domain?
It splits up pretty cleanly…

Naming

Directories

```
FileUtils.copy_directory(…)
FileUtils.create_directory(…)
FileUtils.delete_directory(…)
FileUtils.move_directory(…)
```

Files

```
FileUtils.copy_file(…)
FileUtils.move_file(…)
FileUtils.delete_file(…)
```

Links

```
FileUtils.create_link(…)
FileUtils.delete_link(…)
```

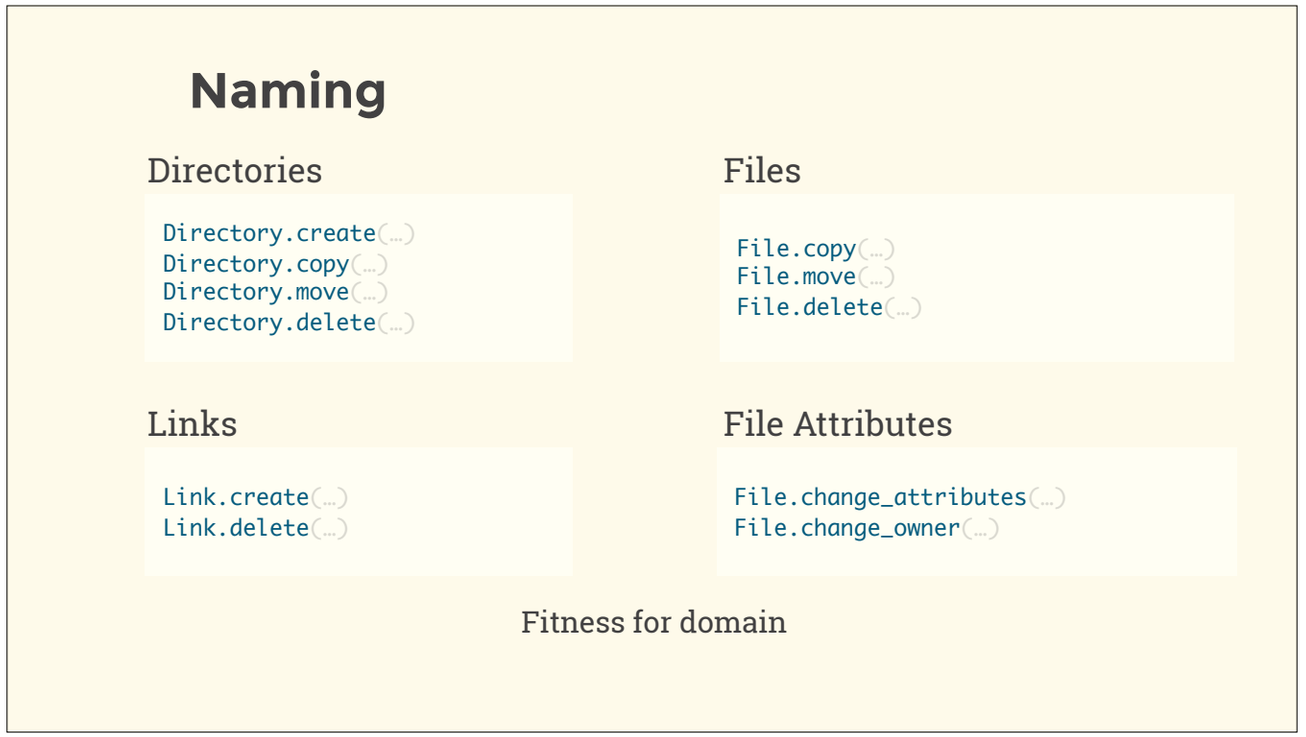
File Attributes

```
FileUtils.change_file_attributes(…)
FileUtils.change_file_owner(…)
```

Fitness for domain

Does that make a better interface?

It *seems* to fit the domain—directories, links & files—without presuming a lot of prior knowledge.



Or we could go further and create separate modules—separate interfaces—for each of these nouns.

… That's interesting, at least, right?

All we did was think about the names of some methods and how we split them up into interfaces.

The question of whether these better fit the domain is a good one.

Naming

```
Directory.new(…)          Link.create(…)
Directory.copy(…)         File.duplicate(…)
Directory.move(…)         File.transmogrify(…)
Directory.delete(…)       Link.destroy(…)
                          File.dismantle(…)
```

Coherence

Coherence in naming is similar, but mostly about consistency…

If you have an operation that brings something into existence, is that “new” or “create”?

Do you “delete”, “destroy”, or “dismantle” it when you’re done?

Being consistent in which names you choose, and applying them broadly…makes it easier for others to understand.

Naming

```
# Sad things

Directory.new_directory(…)

User.process(…)

store_order.user_id  # => “DAL-312348”
employee.user_id     # => “aa7eb462-af83”
```

Descriptiveness

Descriptiveness hinges on word choice and name length. How much context should you include in a single name?
If your interface’s context already includes…like “Directory”, you don’t need…
If your context has a thing called a “User”, make sure the method name makes sense—what does “processing” a user do?
If…already invented…“user id”, don’t invent a different…same name…

Naming

```
POST /authors
GET  /authors/123/articles

POST  /articles
PATCH /articles/321
```

Organization

HTTP interfaces…great chance to think about *organizing* names and interfaces into groupings.
…what goes together from a user's perspective; from use cases.
Well-organized, coherent…easier for brains to understand and absorb.
This will pay dividends when your brain or another brain are trying to understand your interfaces' context in the future. Trust me.

Simple changes have obvious results

Getting back on track, my second observation about worlds that make you feel great was… Simple changes have obvious results. Hopefully this makes some sense just from its description.

Simple changes have obvious results

```
button {
  background: red
}
```

When something in a world is described in a certain way…
…and we attempt to change it…
…we really want the change to be small, isolated, obvious.
Not for a bunch of unrelated things to come crashing down.

Simple changes have obvious results

```
button {
  background: blue
}
```

Isn’t that great?

Simple changes. Obvious results.

Simple changes have obvious results

Some software made for building worlds, like Django, strive to help people create simple, predictable systems like this.
They try hard to ensure…simple changes have obvious results.

Really great documentation

A shortcut: `get_object_or_404()`

It's a very common idiom to use `get()` and raise `Http404` if the object doesn't exist. Django provides a shortcut. Here's the `detail()` view, rewritten:

polls/views.py

```
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

The `get_object_or_404()` function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the `get()` function of the model's manager. It raises `Http404` if the object doesn't exist.

Philosophy

Why do we use a helper function `get_object_or_404()` instead of automatically catching the `ObjectDoesNotExist` exceptions at a higher level, or having the model API raise `Http404` instead of `ObjectDoesNotExist`?

Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling. Some controlled coupling is introduced in the `django.shortcuts` module.

There's also a `get_list_or_404()` function, which works just as `get_object_or_404()` – except using `filter()` instead of `get()`. It raises `Http404` if the list is empty.

Can I take a quick detour?

Talk about great documentation. Django is just…really impressive.

They have a (!) *philosophy* callout in their intro tutorial! Working examples everywhere.

Links…

So great.

Really great documentation

Can I take a quick detour?

Talk about great documentation. Django is just…really impressive.

They have a (!) *philosophy* callout in their intro tutorial! Working examples everywhere.

Links…

So great.

Really great documentation

Stripe too. Really good docs.

Builds confidence in tending the world

…this is all the happy section. We’re in it.

Systems that increase your confidence *while* you work with them will make you a better builder of that world.

Builds confidence in tending the world

- **Small wins add up, especially when learning**
- **Automated tests are confidence builders**
- **Consistency & testability create a positive feedback loop**

Confidence, like trust, is slowly gained and quickly lost.
Use the tools at your disposal to make your worlds safer, more consistent and more confidence-building.

Inspires happiness & empowerment

And don’t forget feelings! Happier developers are more confident developers.

Inspires happiness & empowerment

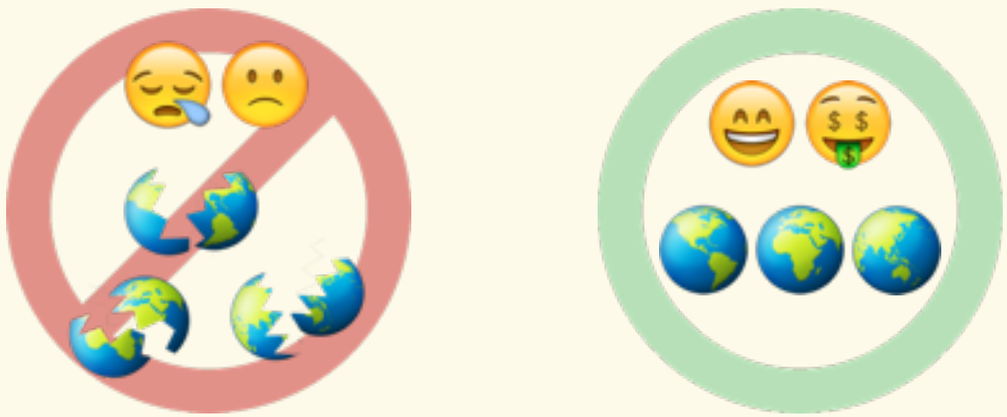
- **Happiness and confidence breed creativity**
- **Empowered developers means nearby worlds improve**
- **The sum of your system's & team's value can grow**
- **You can hire and work with great people!**

Building *good, maintainable, enduring* complex systems—software or otherwise—is an art *and* a science, just like UI design. You *want* creativity.

The spillover effects are great, too: things start to improve outside the lines from where you started.

Several systems get better, your team gets better, your business thrives. You can hire great people.

Inspires happiness & empowerment



I feel a little silly enumerating the benefits of a happy, thriving, confident team.
But there’s an unsavory and persistent faction of our industry invested in burning through code, people and cash.
… Don’t do that. They’ll end up with thousands of tiny, brittle and breaking worlds, wearing down a team of people.
But you’ll have a team who’s building worlds and feeling great! And money too, if that’s your thing.

Takeaways

I hope there’s something there now, in your head, about building worlds and/or feeling great. But I have a few highlights we can review.

Takeaways

- **Systems have lots of interfaces**
- **Take the time to think about them (give things good names!)**
- **Create your worlds from small, great, consistent stories**
- **Humans first. Humans first. Humans first. Have empathy.**

Interfaces. They’re important. Look for them everywhere and think…
…when creating new ones, think for 5 min about names of things (methods, URLs, fields).
Think about systems in small pieces, compose them together and couple them loosely.
I can’t stress…Before your fingers fall onto your keyboard, think about the other minds building your world with you.

Bit Operations

X	Y	X&Y	X|Y	X^Y	~X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Also, one last thing. Please don’t ever use bit operations, bit shifting or bit masks.
It’s this classic thing where people think about how efficient and clever they are. So they find some problem…
…and they think: “Hey, I’ll use bit operations!” (!)

Bit Operations

X	Y	X&Y	X|Y	X^Y	~X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

...and now they have

(11101001 ^ 00101011) | (01010111 & 10111000)) << 32

problems...

Also, one last thing. Please don’t ever use bit operations, bit shifting or bit masks.
It’s this classic thing where people think about how efficient and clever they are. So they find some problem…
…and they think: “Hey, I’ll use bit operations!” (!)

Thank You

Thank you.

Credits & Colophon

- “Brain” by 녕진소 from the Noun Project, purchased for use
- “B-52 lower deck” from Wikimedia Commons by Desertsky85450 [Public Domain]
- “Patravi” watch from Wikimedia Commons by Carlfbucherer [CC BY-SA 4.0]
- Typefaces are Montserrat & Roboto Slab from Google Fonts [Open Font License]
- Color palette is “Adrift in Dreams” by Skyblue2u on Colour Lovers
- Diagrams built with OmniGraffle & Keynote, images edited in Pixelmator

- This slide deck is released under a Creative Commons BY-SA 4.0 License.

Made in Austin, Texas · July–August, 2016

https://interfaces.design/