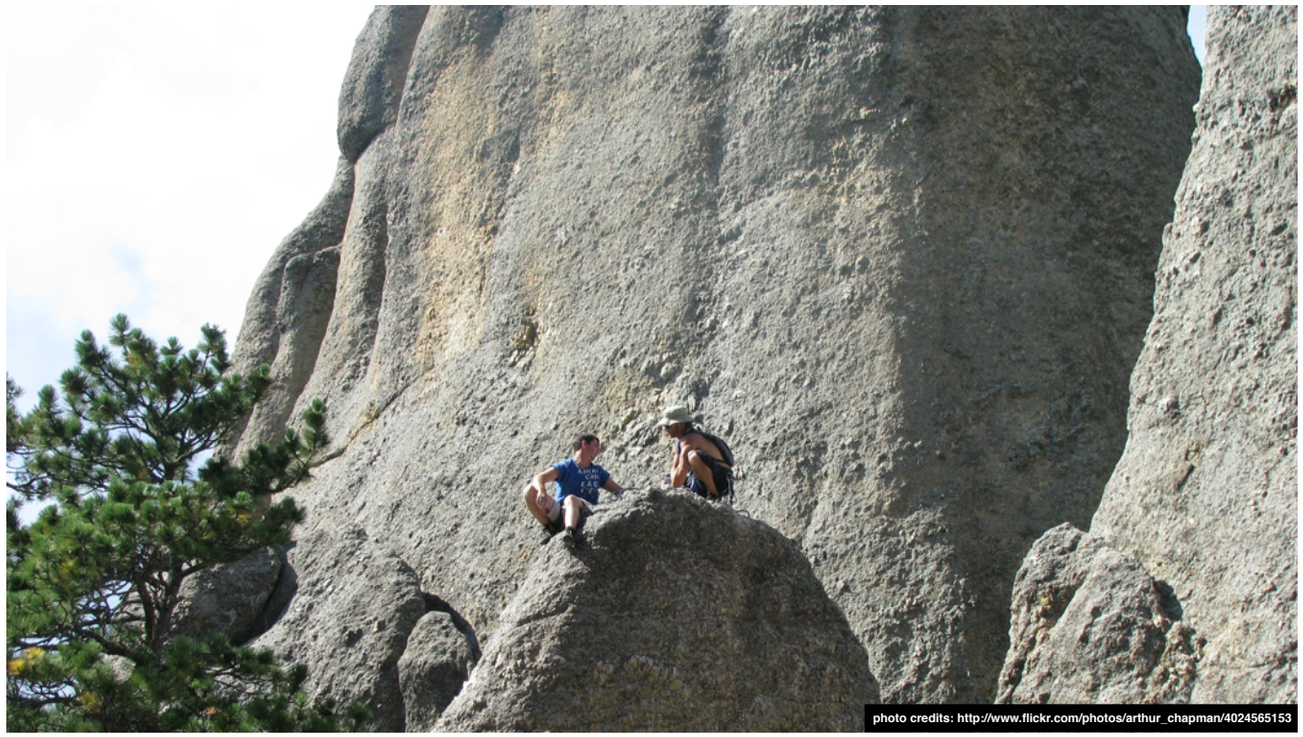
S.O.L.I.D. JavaScript

In A Wobbly World (wide web)


photo credits: http://www.flickr.com/photos/gfoster67/7356180472

pillars of software design
typically in object oriented software: polymorphism, inheritance, encapsulation
still apply to javascript, but implemented differently



most people misunderstand these pillars
make it more difficult than it needs to be
as if they were free-climbing a mountain - no guides, no safety nets

this leads to train-wreck projects
code that falls apart under load
immoveable and inflexible, because we tried to build mountains to climb
there is a better and easier way

build a staircase to get us to the top
break down the three pillars in to multiple steps
reinforce the stairs and evolve them as needed

S.O.L.I.D.

this is where the SOLID principles come in
because software development should not be a jenga game
an acronym of acronyms
not the only principles to follow, but one place to start

S.O.L.I.D.


SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

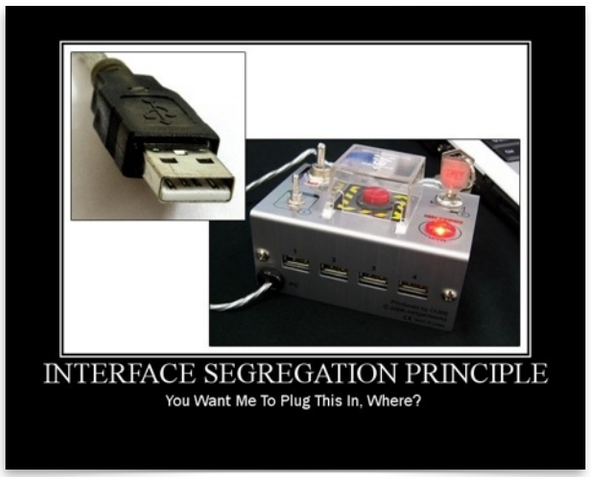
S.**O**.L.I.D.


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

S.O.**L**.I.D.


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

S.O.L.I.D.


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

S.O.L.I.**D**.


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

dive in to the principles
see how they can apply to JavaScript
has object oriented capabilities
very different implementations than C/C#/Java - where the SOLID principles came from

Single Responsibility

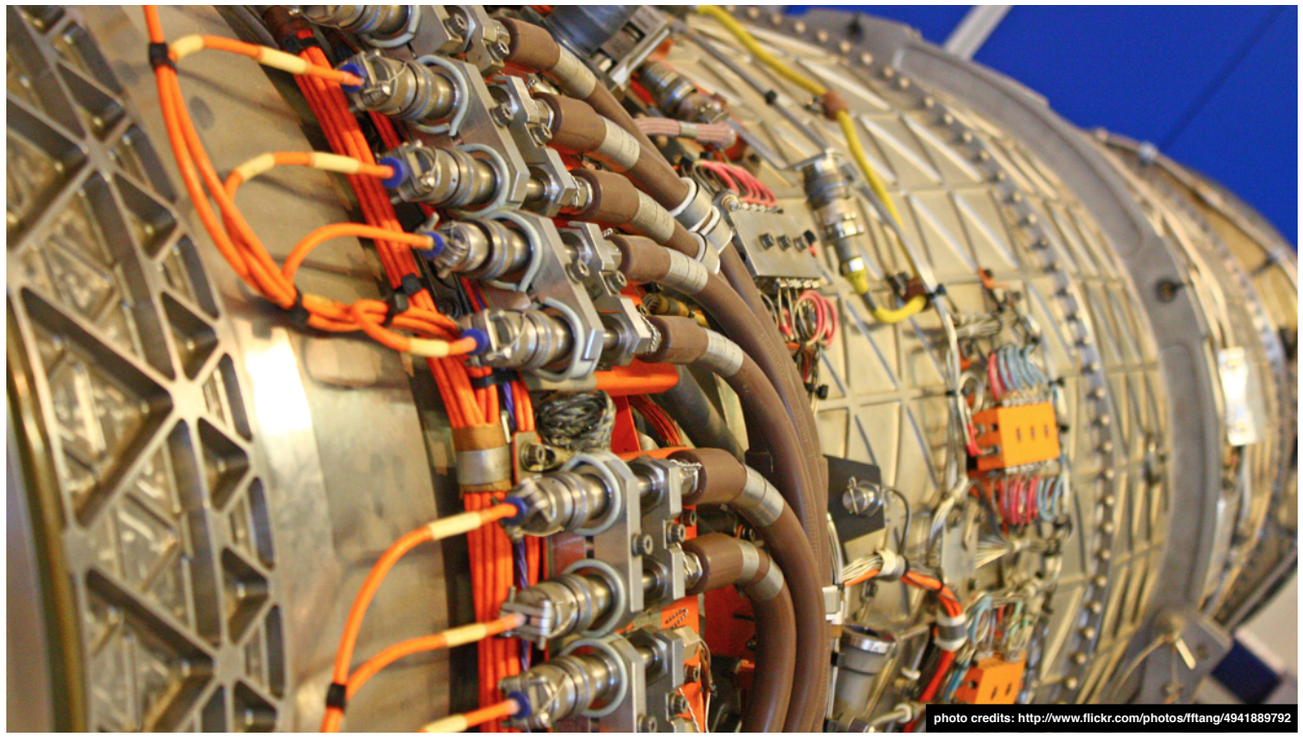
~~every [object, module, etc] should have one reason to exist~~

single responsibility
one reason to exist… one responsibility per thing
… is the wrong definition

Single Responsibility

every [object, module, etc] should have
one, and only one, reason to change

one reason to change
difference is subtle, but important
it's a matter of perspective
but that perspective changes the approach to code



example: are you building a jet?

or are you building a jet?

building or modeling a jet turbine for an engine maintenance system requires infinitely more detailed specs and object and design than building a lego jet.

if you don’t understand what you are building, you won’t build the right abstractions and object graphs

```
$(“#code”).show();
```

show code… ask if breaking SRP
maybe - can’t know this without having a higher level understanding of what you’re building, and why

tell me the shape of this forest
show me how to get where i'm going, from here

you need to know “the big picture”
to know where you are, currently, where you’re heading, and why

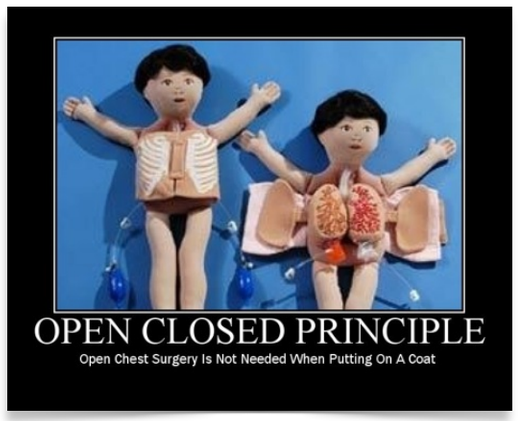
```
$(“#code”).show();
```

show workflow example, with big picture and detail mixed together
break the workflow apart in to separate object
each resulting object has one reason to change, understanding the larger workflow organizes that reason

Open-Closed

Open to extension, closed to modification

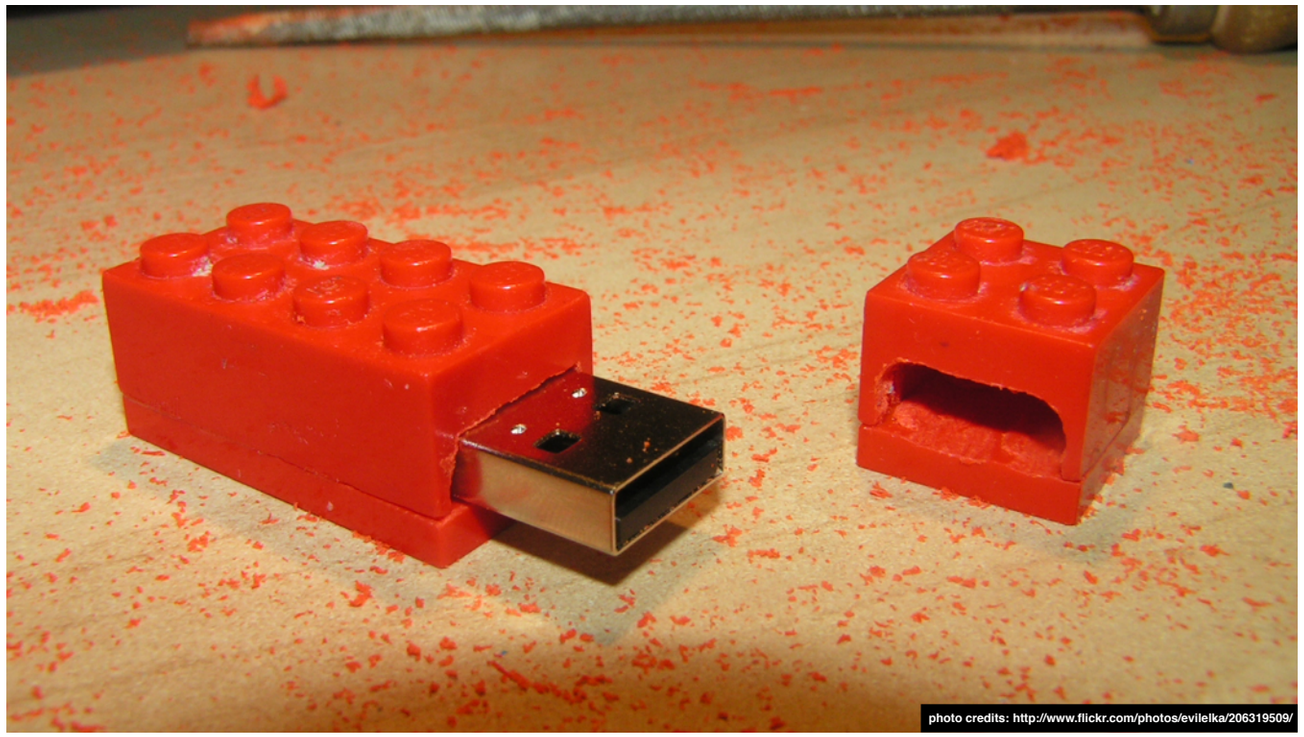
Extend an object's behavior, without modifying the object



change behavior without changing code?

photo credits: http://www.flickr.com/photos/iguanajo/40048833/

how is that possible?
how can something be open to change and closed to change?
what does it mean?



you've seen this in real life… USB, legos, tinker-toys, etc
it's all about plug & play…
a printer, a camera, a scanner, a hard drive, a headset with mic
modify the capabilities and behavior of a computer
no change to the internals of that computer
OCP is the ability to plug new behavior in to known extension points

```
$(“#code”).show();
```

show code w/ switch statement
refactor it to a registry object
show how registry allows easier extension

Liskov Substitution

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T

math formula to explain type substitution
a bit esoteric

Liskov Substitution

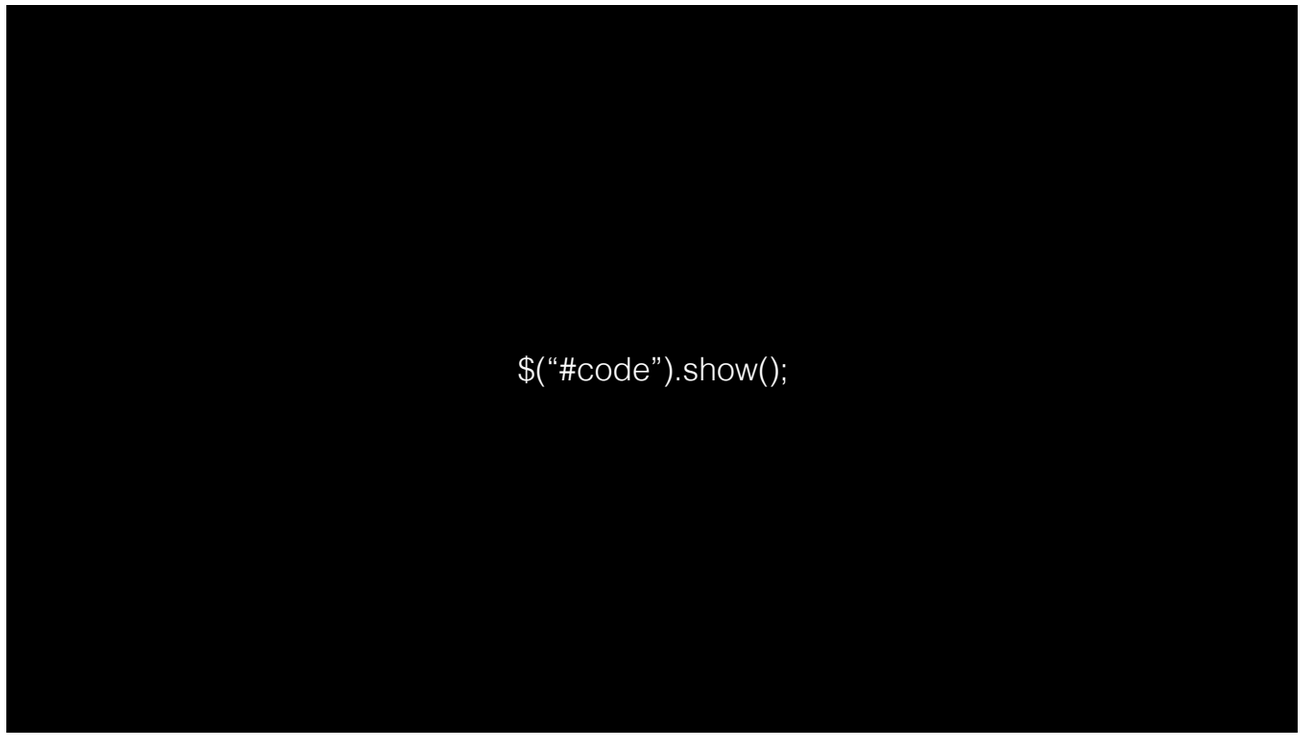
Derived [objects / types] must
be substitutable for their base

polymorphism, apis
with semantics and meaning

photo credits: http://www.flickr.com/photos/sfllaw/222795669

take one thing out and put another one in
are you sure the new thing fits properly?

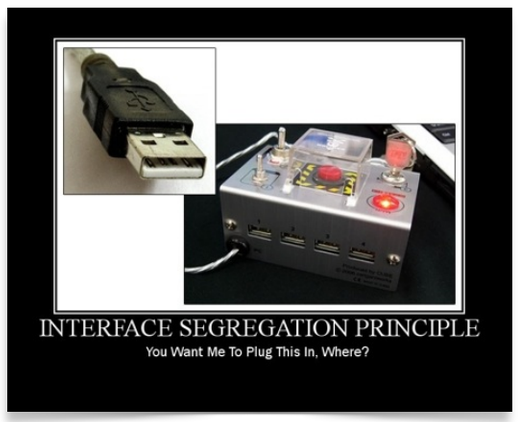
semantics
squares and rectangles



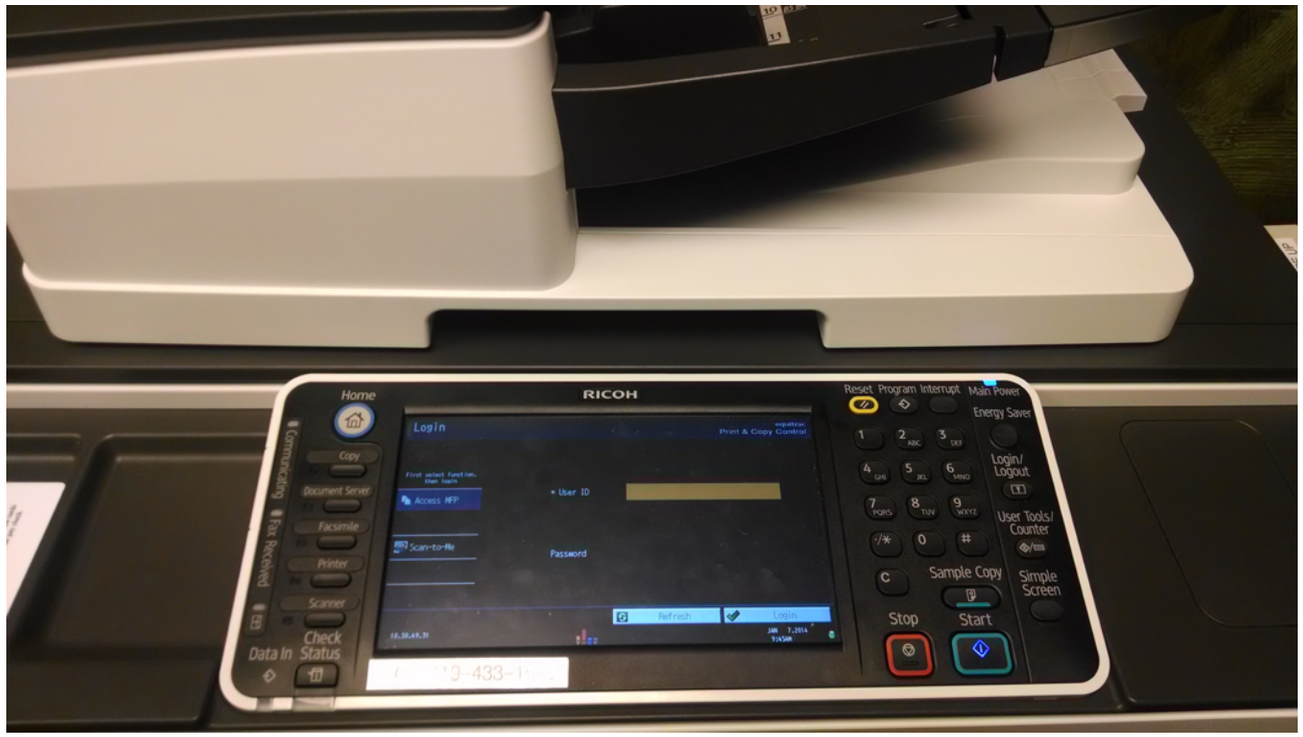
show the square vs rectangle code
squares are not rectangles in code because of semantic violations
would be better to have a shape object to inherit from
brings up another problem, though - prototypal inheritance and not interface constructs in javascript

Interface Segregation

Make fine grained interfaces
that are client specific



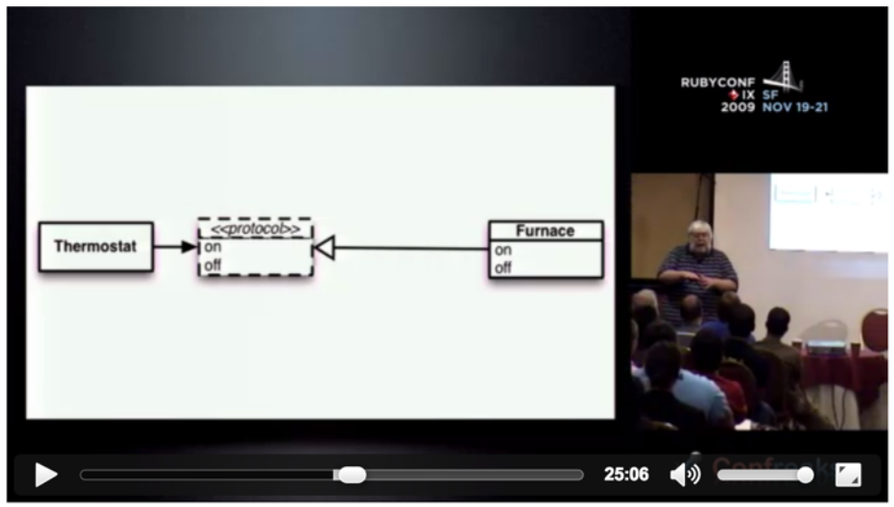
interface === API… more on that in a minute



multi-function printer
too many buttons, knobs, options
too much effort to understand it and use it correctly

just give me this!
simple, easy, one button is all i need
… but interfaces in javascript?

Ruby Conference 2009



SOLID Ruby

Jim Weirich

use of protocols instead of explicit interfaces
more like a handshake agreement vs a contract
relies on documentation more

```
$(“#code”).show();
```

show c# example of interface declaration
talk about how javascript doesn’t have this
show the inheritance example and how it’s not needed
simplify the code with a shape “protocol”
keep it simple, specific to working with a shape
goes back to single responsibility - a different perspective on the same ideas.
SRP is often internals and behaviors, while ISP is external API access to the behavior

Dependency Inversion

Two parts to this principle:
abstraction and ownership

Dependency Inversion

Depend on abstractions,
not on concrete things

1st half… abstraction

changes in code have a ripple effect
need to control the ripple - the direction it moves, how far out it goes
introduce abstractions (protocols from ISP)
inject them in to the object
the thing that uses the protocol is not in control of creating the thing that implements it - needs fewer of those details

software should be able to ripple when it needs to
should not be immovable lava once it settles

```
$(“#code”).show();
```

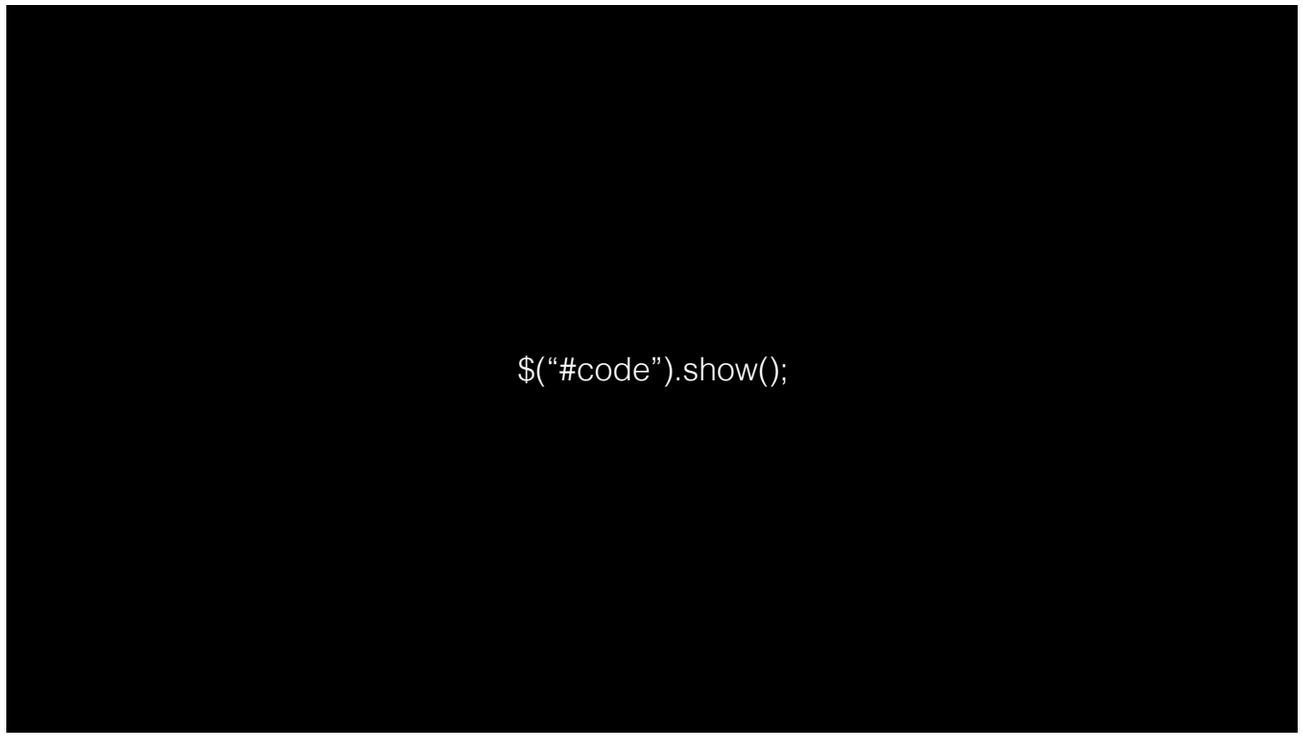
show the hard coded constructor here
extract it in to a dependency being passed in to the constructor
this is only half of the principle
but this is where most people leave it

Dependency Inversion

Detail depends on policy.
Policy does not depend on detail.

2nd half… ownership

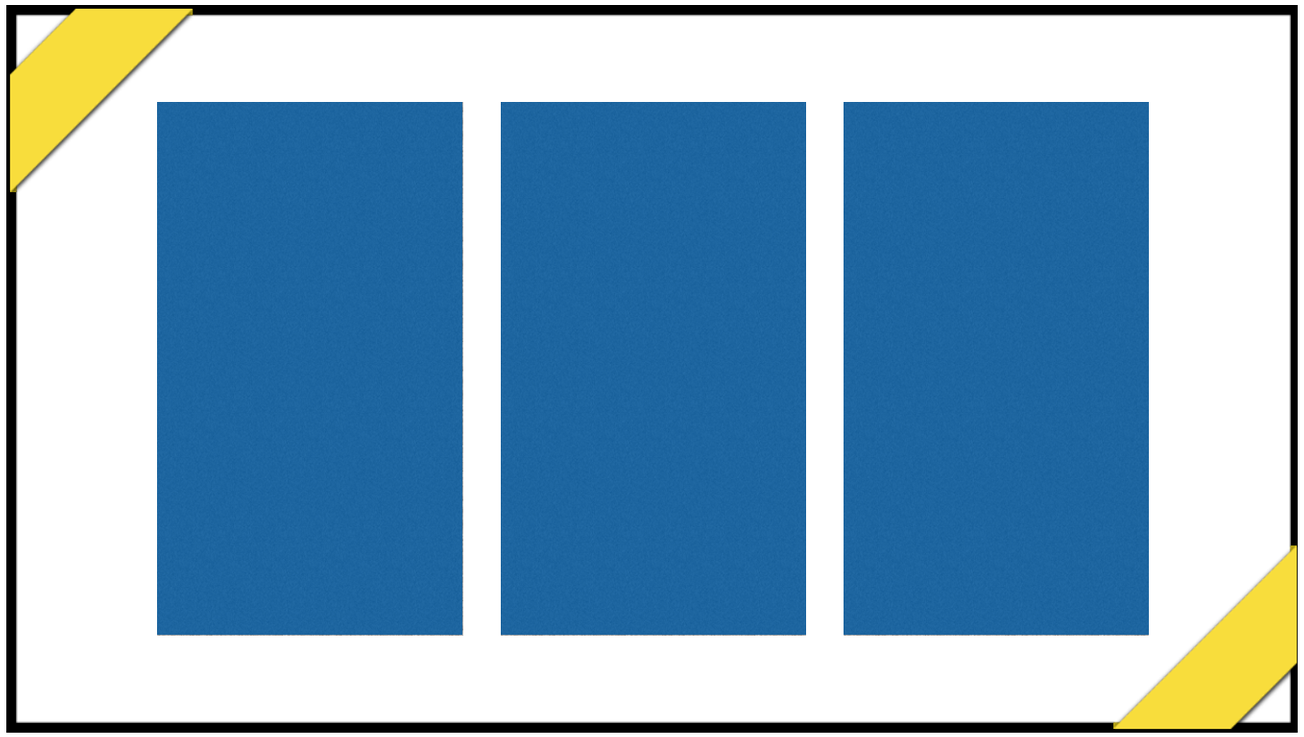
does the key own the lock?
do you grab a key and find the lock it fit?
or does the lock own the key?
you find the key that fits the lock you need to open or close



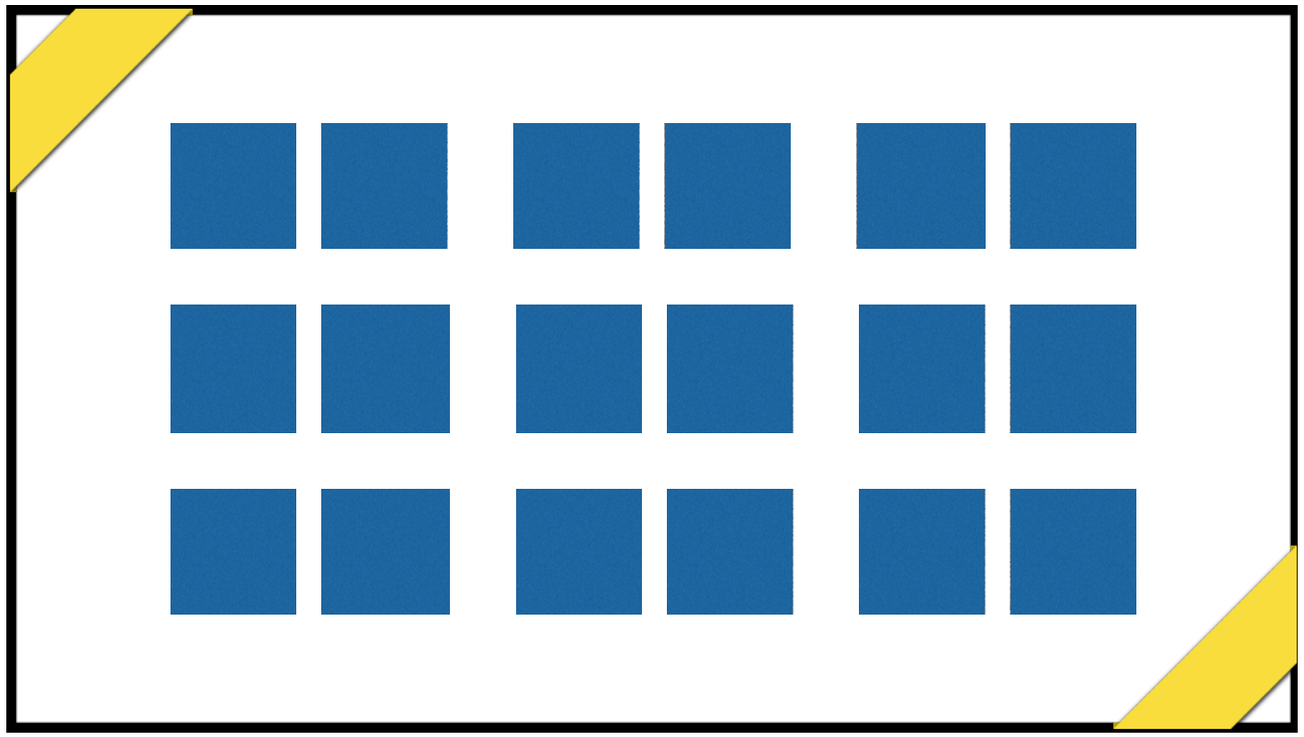
show the lock and key code
explain the protocol
show how the method names are like the tumblers
how the key must conform to the lock’s specifications, or it breaks
the lock owns the key’s specs.
the detail (the key) depends on the policy (the lock)
further reduce the ripples of code changes by knowing who owns the abstraction / protocol

What Did We Gain?

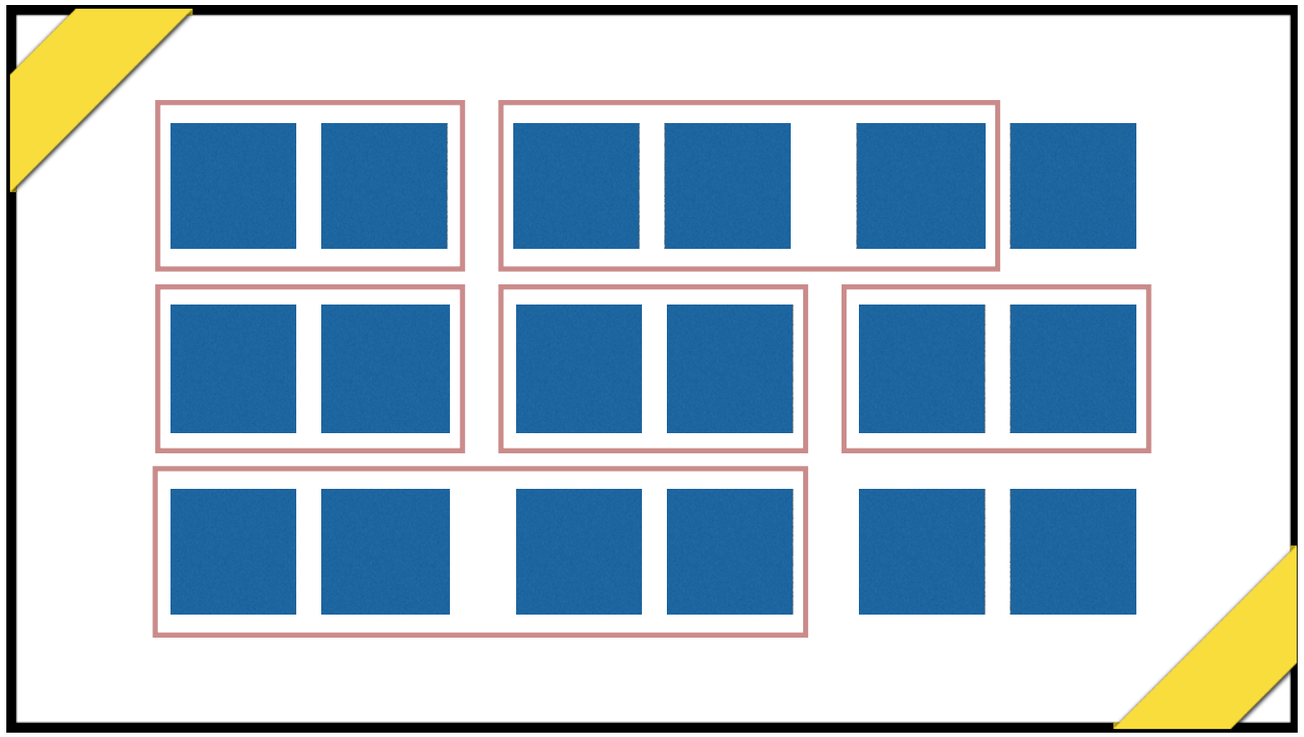
Are there any drawbacks?



large, monolithic code
difficult to change
easy to break
hard to understand



broken down in to many smaller things
easier to understand each of those things
but it does mean more things to keep track of, which can feel daunting



natural relationships appear
group the code by these relationships
split up responsibilities by teams or sub-teams
makes it easier to look at the larger picture of each group
dive in to the detail of the individual group when needed

burn your jenga game

build solid code, with a strong foundation, on the pillars of principle driven software development

flow like water, using SOLID as a starting point

SOLID are not the only principles that matter

A Bit About This Guy

Developer Advocate for Kendo UI

- KendoUI.com @kendoui

Blogger, Screencaster, SaaS-er, etc.

- derickbailey@gmail.com
- DerickBailey.LosTechies.com
  @derickbailey
- WatchMeCode.net @watchmecode
- SignalLeaf.com @signalleaf