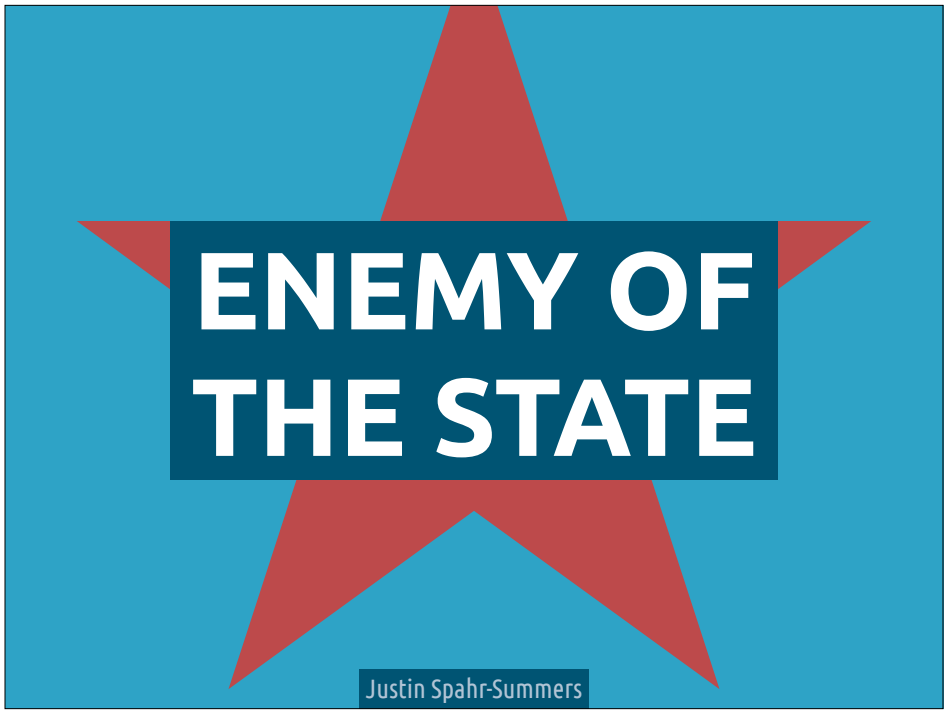
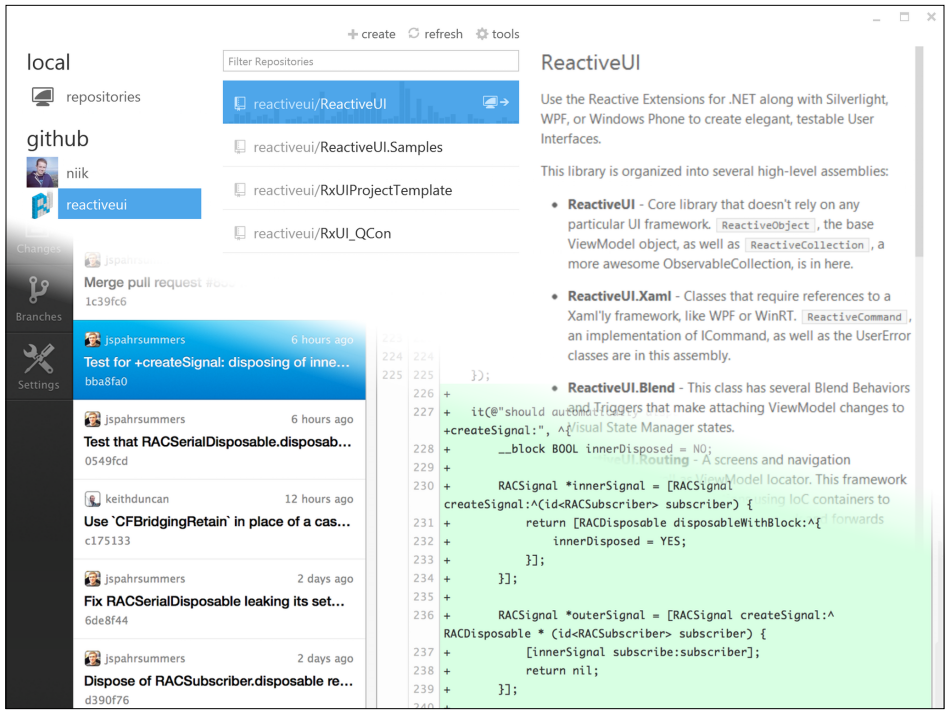

ENEMY OF
THE STATE
Justin Spahr-Summers

First of all, let me introduce myself. My name is Justin Spahr-Summers, and I'm a desktop developer at GitHub.



I've mostly worked on GitHub for Mac, but I've recently started contributing to GitHub for Windows as well. Basically everything I'm gonna talk about today applies to both.

I could've gotten up here to talk about any number of concrete things, like Mantle (a model framework we created), or how we build GitHub for Mac, or even The GitHub Flow™.

However, I really want to impart some more abstract knowledge, so this may be a less concrete talk than what you're used to. Don't let your eyes glaze over yet, though, because programming is *all about* abstraction, and—despite what you may have heard—an understanding of theory is *hugely important* for solving practical, real world problems.

In other words, I don't want to teach you how to use one concrete thing. My goal is that everyone walks away from this talk *a better programmer*.

What even is state?

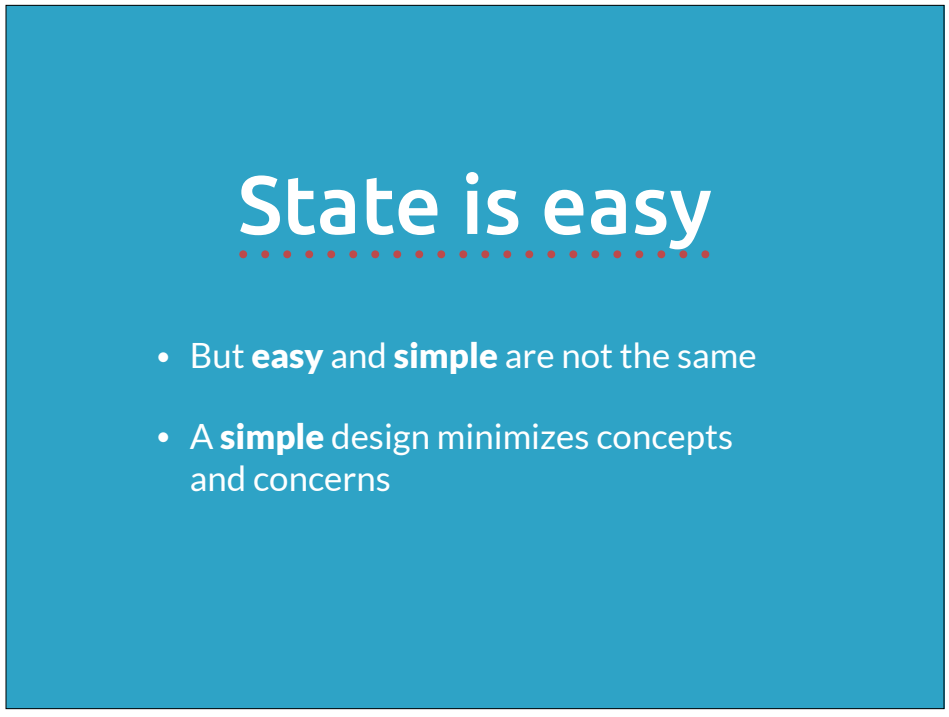
- **State** refers to a program's stored values at any given time
- **Mutation** is the act of updating some state in-place

A "value" is unchanging, it has no concept of time. "State" consists of the values you currently have at once.

Variables are state

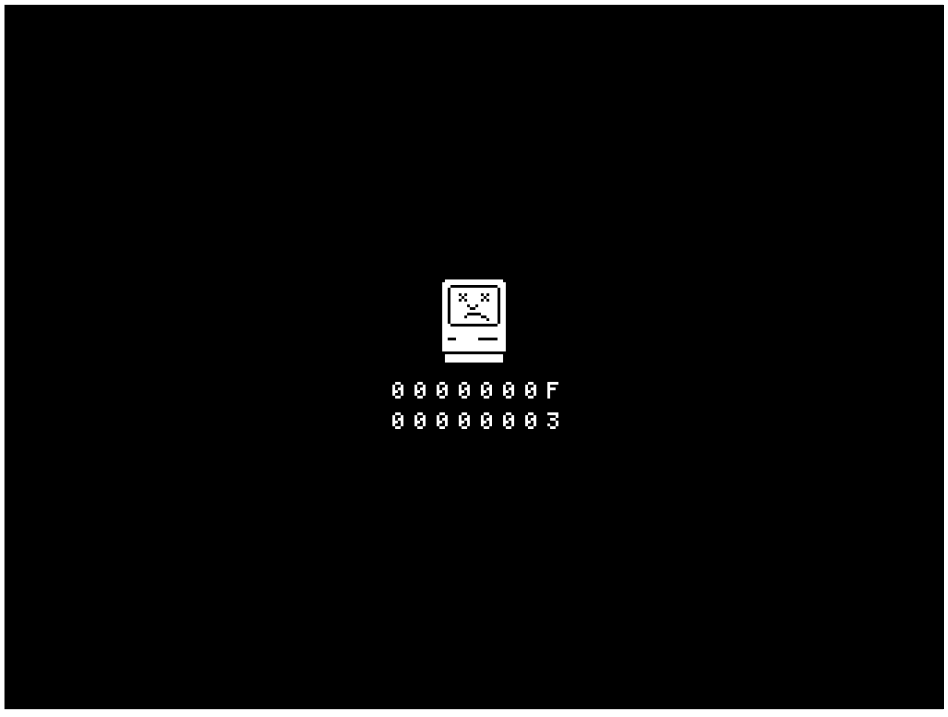
```
int a;

// Store a value
a = 5;

// Update (mutate) the value
a = 2;

// Mutate the value again
a = a + 1;
```

Variables are the most obvious kind of state in any program. When you want to hold on to a new value, you change the variable in-place (you *mutate* it).



State is easy because it's *familiar*, or approachable. We first learned how to program in a stateful way, so it comes naturally to us… but that doesn't mean it's right.

See Rich Hickey's "Simple Made Easy" talk for more about “simple” vs. “easy.”

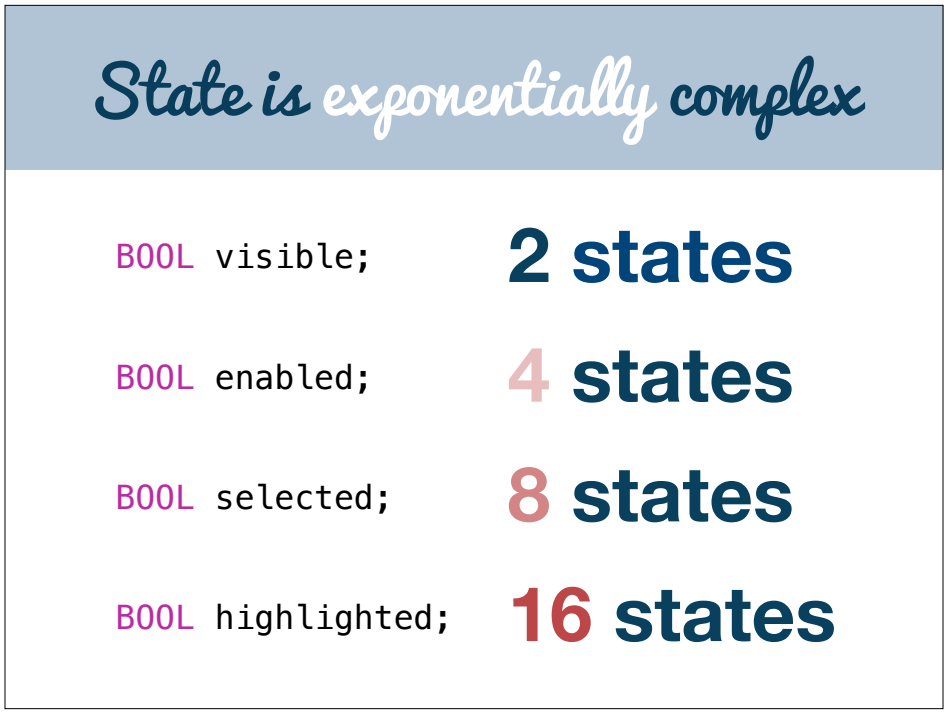


Oops! One huge problem with state is that it can "go bad." Any time you've ever restarted a computer or an app, and it fixed an issue you were having, you've been a victim of state.

For example, maybe two different code paths updated some state simultaneously. If the results are inconsistent, the program (or even operating system) can crash.

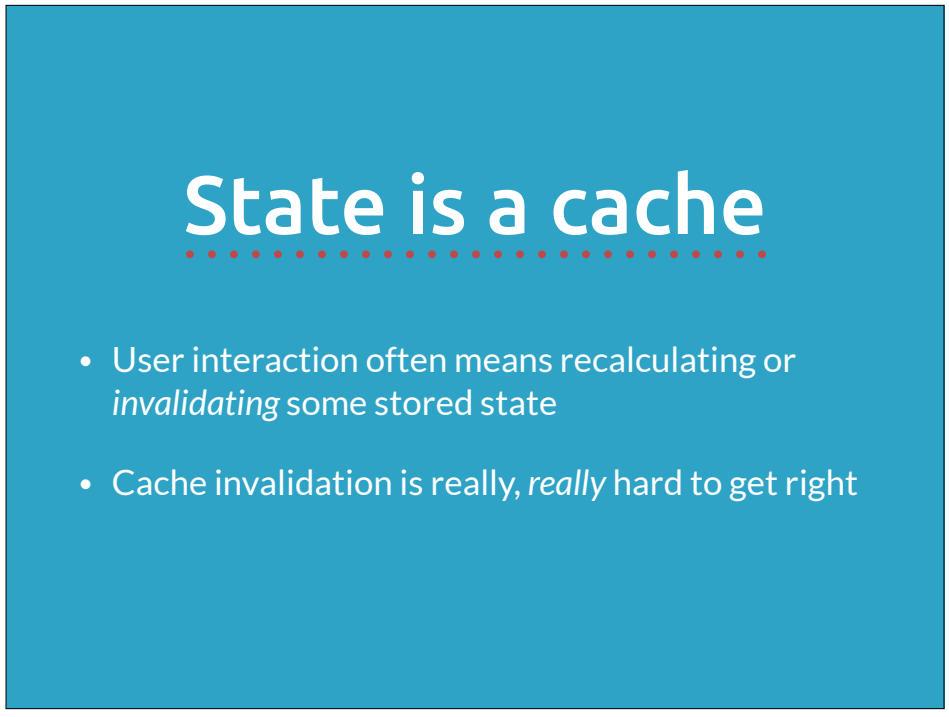
Complexity arises from multiple concepts or concerns being interleaved. State complicates everything it touches.

See Moseley and Marks' "Out of the Tar Pit" paper.



To illustrate the complexity of state: as you add each new boolean, you *double* the total number of states that your class can be in. For more complicated data types, the growth in complexity is even more dramatic.

Needless to say, this problem is greatly exacerbated with *global* state, because it increases the complexity of your *whole program* exponentially.



This is true every time any state is *aliased*, or stored in more than one location. For example, if you have a text field with some content, and then also track some version of that content in your view controller, one of those is effectively a cache for the other.

See Andy Matuschak's Quora post, "Mutability, aliasing, and the caches you didn't know you had."

Example: Table view updates

```
[self.models removeLastObject];

NSIndexPath *indexPath = [NSIndexPath
  indexPathForRow:self.models.count
        inSection:0];

// Whoops, inconsistency! 💥
[self.tableView
  insertRowsAtIndexPaths:@[ indexPath ]
        withRowAnimation:animation];
```

UITableView maintains a cache of its rows and sections, and is only "invalidated" by sending the proper delete/insertion messages to the view. If you send messages that are inconsistent with your model, you're gonna crash.

This code is obviously contrived, but it's pretty easy to run into less obvious cases in the Real World™. A less stateful design would have the table view automatically reacting to an external source of data (kinda like bindings), so its rows could never become inconsistent with the data.

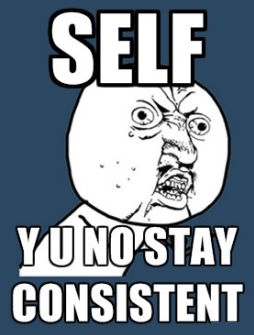
State is nondeterministic

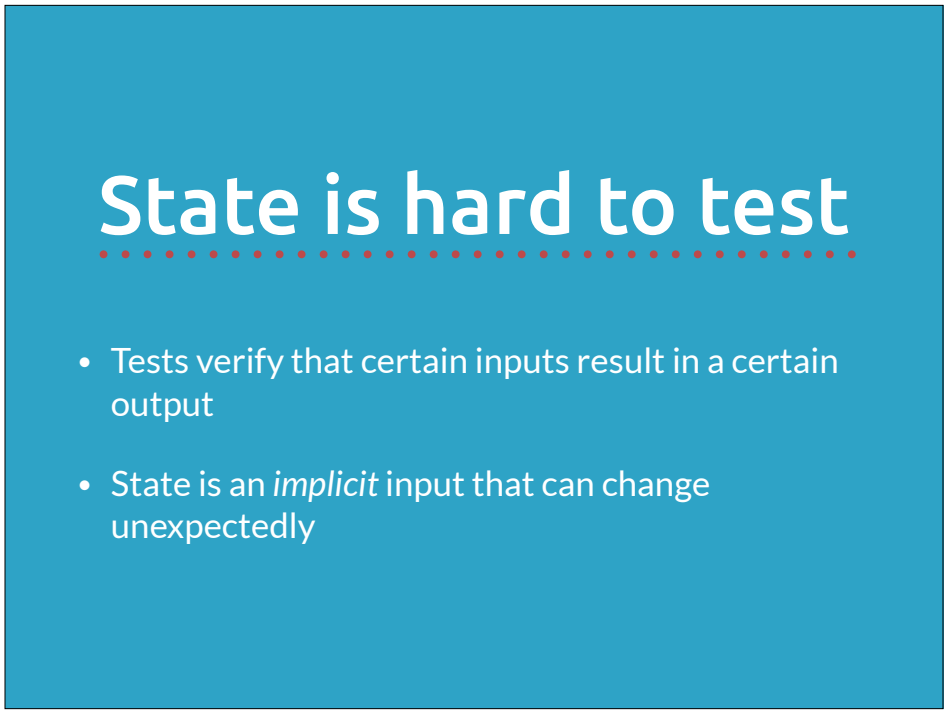
- Race conditions can result in corruption or inconsistent state
- Variables can change unpredictably, making code difficult to reason about

For example, if two threads update a variable almost simultaneously, what's the result? One update "wins" and we don't get to do anything with the other.

State is nondeterministic

```
int x = self.myInt;

NSLog(@"%i", x);
==> 5

int y = self.myInt;

NSLog(@"%i", y);
==> 10
```





Not only is it complicated to set up a correct initial state for testing, but method calls can change it *during* the test, which can introduce issues with ordering and repeatability.

By contrast, it's much easier to test a pure algorithm, where the output is only determined by its explicit inputs.

Example: Testing Core Data

```
id managedObject = [OCMockObject mockForClass:[NSManagedObject class]];
id context = [OCMockObject
  mockForClass:[NSManagedObjectContext class]];

[[context expect] deleteObject:managedObject];
[[[context stub] andReturnValue:@YES] save:[OCMArg anyObjectRef]];

id resultsController = [OCMockObject
  mockForClass:[NSFetchedResultsController class]];

[[[resultsController stub] andReturn:context] managedObjectContext];
[[[resultsController stub] andReturn:managedObject]
  objectAtIndexPath:OCMOCK_ANY];

id viewController = partialMockForViewController();
[[[viewController stub] andReturn:resultsController]
  fetchedResultsController];

[viewController deleteObjectAtIndexPath:nil];
[context verify];
```

problem?

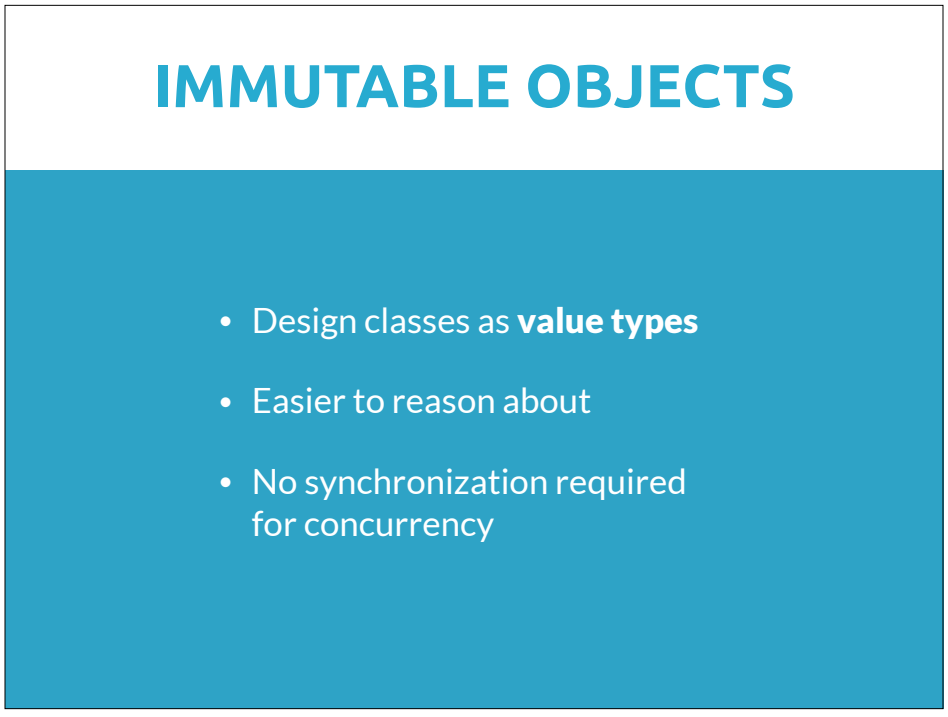
This is a slightly modified example from Ash Furrow's C-41 project, testing that a view controller's NSFetchedResultsController successfully updates after a managed object is deleted from the context.

As you can see, it uses a lot of mocks and stubs to avoid *actually* manipulating a database (which is a form of state). Stateless code requires less mocking and stubbing, since the output of a method should only depend on its input!

Most applications require some state, and *that's okay*. Here are some examples of state being necessary and helpful for solving a particular problem.

[Talk about each one individually]

Although it's not possible to eliminate *all* state from a Cocoa application, we can try to minimize it (and therefore minimize complexity) as much as possible.

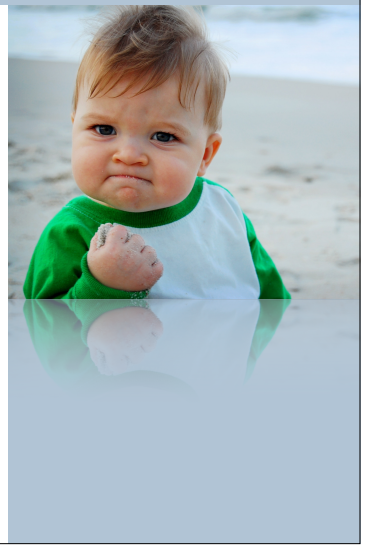


Since an immutable object (by definition) cannot change, it's a great way to eliminate stateful behavior.

We can stop thinking about objects as "things to store and manipulate data on," and start thinking about them as a convenient way to structure *immutable* values. NSNumber, NSValue, NSDate, and the immutable collections of Cocoa are all great examples.

IMMUTABLE OBJECTS

```
int x = num.intValue;

NSLog(@"%i", x);
==> 5

int y = num.intValue;

NSLog(@"%i", y);
==> 5
```



If `num` is immutable here, there's no way its properties can change between reads. It's always *consistent*.

CONSISTENCY

```
// What happens if the Department object
// is used between these two lines?

department.employees = newEmployees;

department.selectedEmployee =
  newEmployees[0];
```

Some properties need to be consistent with each other (for example, a list of employees, and the currently selected employee). When mutating an object in-place, it's easy for them to fall out of sync, or for the object to be used before both properties have been updated.

Immutable objects can be validated once, and holistically, because all values are set at the same time (initialization). There will never be changes after initialization, so immutable objects can be designed to make inconsistent/invalid states actually *impossible*. If you pass in inconsistent data, just return *nil* and fail to initialize.

TRANSFORMATION

```
// Names are “updated” by transforming
// a copy and using that going forward.
NSString *newName = [name
  stringByAppendingString:honorific];

NSArray *newNames = [self.names
  arrayByAddingObject:newName];

// `self` is being mutated, but the array
// itself is not.
self.names = newNames;
```

Of course, even when objects are immutable, you still want some way to change them. Most kinds of mutation can be replaced with *transformation*, which involves copying the original while simultaneously making a change. NSString and NSArray have a few handy methods for this already.

This last line bears some explaining. We're still mutating `self` by changing one of its variables, but the *array itself* is never updated in-place. If something else has read the previous value of `self.names`, that copy is still valid and has not changed. This is a form of *isolation* (to be continued).

NSArray is a good example for this too. Instead of using -arrayByAddingObject:, you can create a *mutable* copy of the array, change that copy in-place however you want, then convert it back into an immutable array.

CLASS CLUSTERS

```
NSMutableArray *newNames = [self.names
  mutableCopy];

[newNames removeLastObject];
[newNames insertObject:newName
               atIndex:0];

self.names = [newNames copy];
```

The final copy (from mutable to immutable) here isn't strictly necessary, as long as the array is never mutated again, but I consider it to be good practice. You can (and should) get the same effect by marking the property with the `copy` attribute.

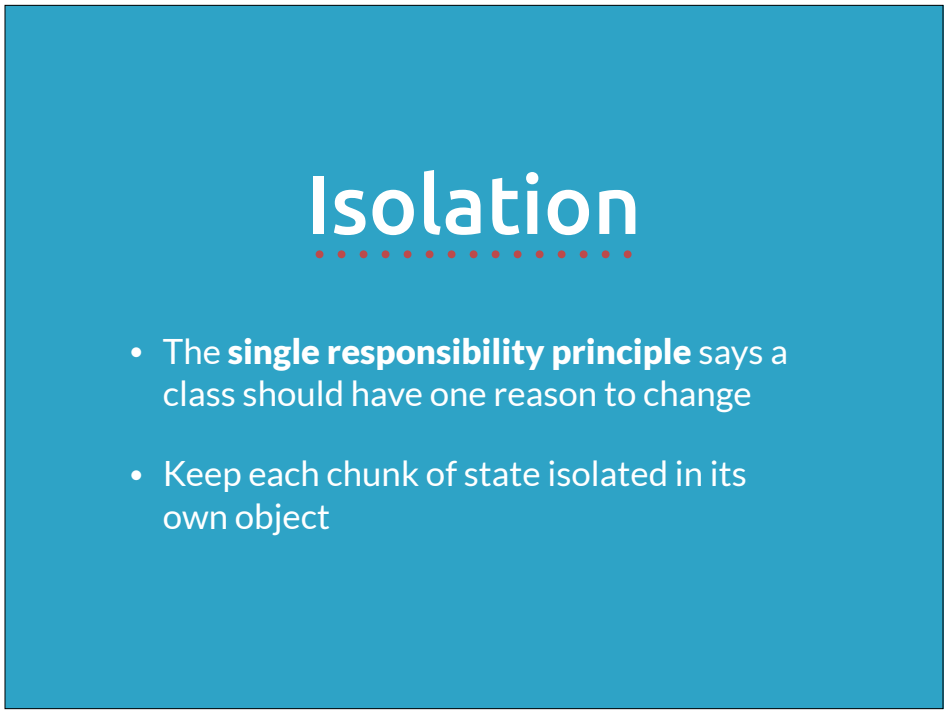
LIGHTWEIGHT IMMUTABILITY

```
JSArray *newNames = [self.names
  update:^(id<JSMutableArray> names) {
    [names removeLastObject];
    [names insertObject:newName
                atIndex:0];
  }];

self.names = newNames;
```

We can even tighten up the class cluster approach a bit, like in this adapted example from Jon Sterling's blog post, "A Pattern for Lightweight Immutability in Objective-C." This way, all mutation is kept confined to a small `update:` block. The returned array is immutable, and has been completely transformed in one fell swoop.

Here, we still get all the benefits of immutable objects, but an easy way to transform them when needed.

When stateful changes can't be replaced with immutable transformations, we should do our best to reduce its impact.

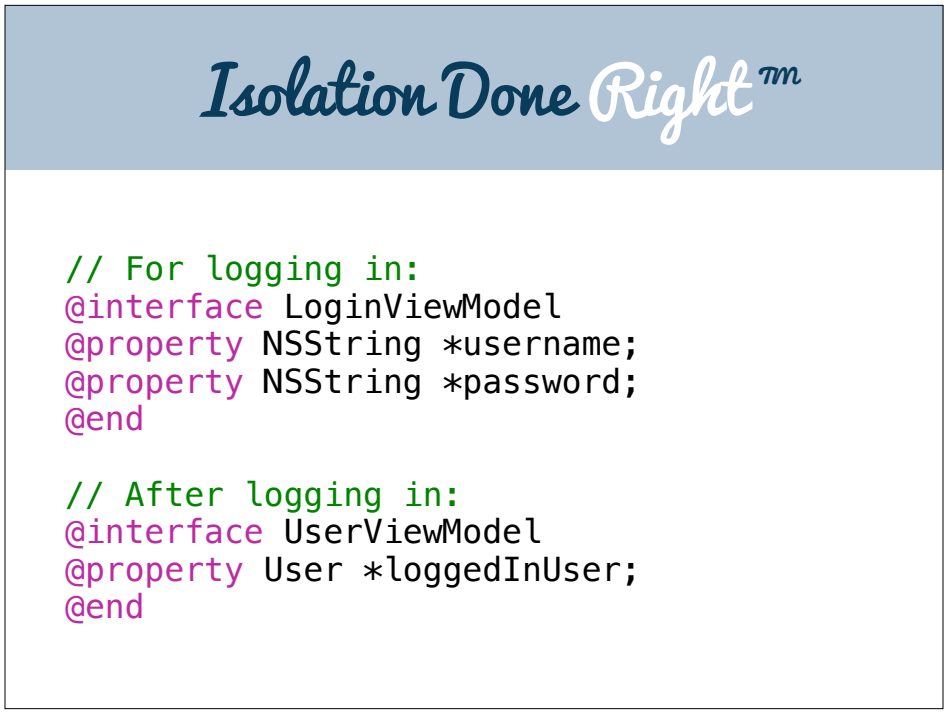


The most effective way to simplify state *without removing it* is to isolate it, so it doesn't get complected with other concerns.

Isolation Done Wrong™

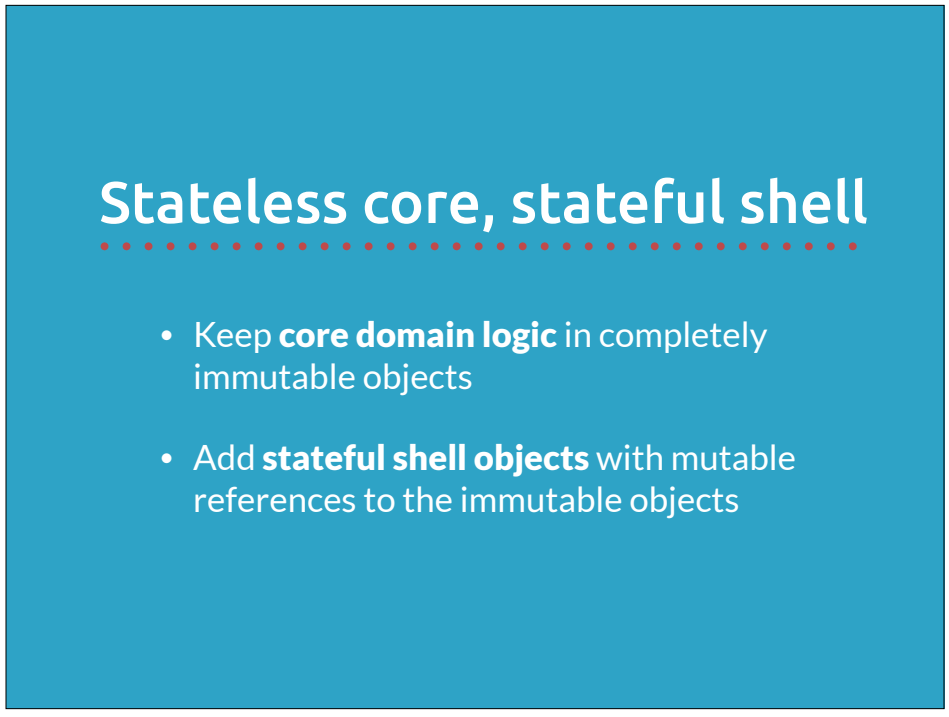
```
@interface MyViewController

// For logging in:
@property NSString *username;
@property NSString *password;

// After logging in:
@property User *loggedInUser;

@end
```

This is an example of poor isolation. The view controller is "complecting" the concern of *logging in* with the concern of *knowing who's logged in*. As the implementation grows to manage both of these concerns, it becomes difficult to reason about them separately.



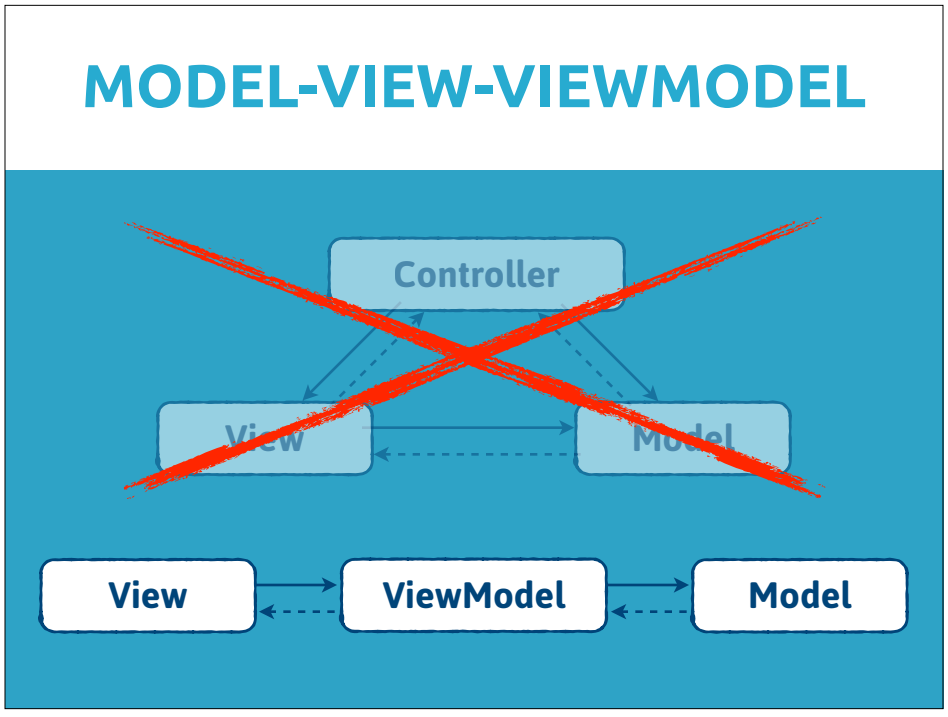
By splitting these two concerns out into separate objects, there's less of an opportunity for them to interact—and, if they do, it'll at least be more explicit.

Furthermore, the specific states (in the form of these "view models") can now be passed to methods that operate upon them. There's less confusion about the inputs to a method when the state is more limited and explicit.



Basically, use immutable objects and pure algorithms for as much as possible. Once you get to the point where you need some state, wrap it *around* the immutable stuff. Your stateful shell can transform the immutable objects and update references to them.

See Gary Bernhardt's "Boundaries" talk.



Model-View-ViewModel is actually a great example of this "stateless core" design. MVVM (depicted here, on the bottom) involves replacing the omniscient controller of MVC with a less ambitious "view model" object. The view model is actually owned by the view, and behaves like an adapter of the model.

The view model is typically responsible for changing the model, which means you can make the model immutable, and the VM can just apply transformations instead of mutations.

MODEL-VIEW-VIEWMODEL

```
@implementation UserViewModel

- (void)loggedInWithUsername:
    (NSString *)username
{
  self.user = [self.user
    userByReplacingUsername:username];
}

@end
```

Here's a completely contrived example, using the UserViewModel and its User property from before. Even if the User class is immutable, the view model can still update its `user` property by *transforming* the model and keeping the new version. The User is the stateless core, and the UserViewModel is the stateful shell.

If you're interested in learning more about MVVM, see the ReactiveCocoa/ReactiveViewModel repository.

Globals: Just Say No

- **Singletons** are global state
- Any global state gets mixed in to *every* object in the program
- Pass instances around instead

Maintaining some global state is effectively like adding that state to *every object* in your codebase.

If isolation reduces complexity, global state compounds it. It's the exact *opposite* of good isolation.

In a move that will not surprise anyone who knows me, I'm now going to talk about functional reactive programming a bit.

ReactiveCocoa is our implementation of Functional Reactive Programming. FRP is a completely different approach to managing state, where in-place changes are replaced by streams of values known as "signals."

Values over time

```
int x;

x = 1;
x = 2;
x = 3;
…
```

The huge benefit to this approach is that *time* is now a first-class concern. Unlike a variable, where the past values are lost forever, it becomes possible to manipulate *all* of a signal's values—past and future.

Assignment over time

```
RAC(self, name) = [field.rac_textSignal
  filter:^ BOOL (NSString *name) {
    return name.length > 0;
  }];
```

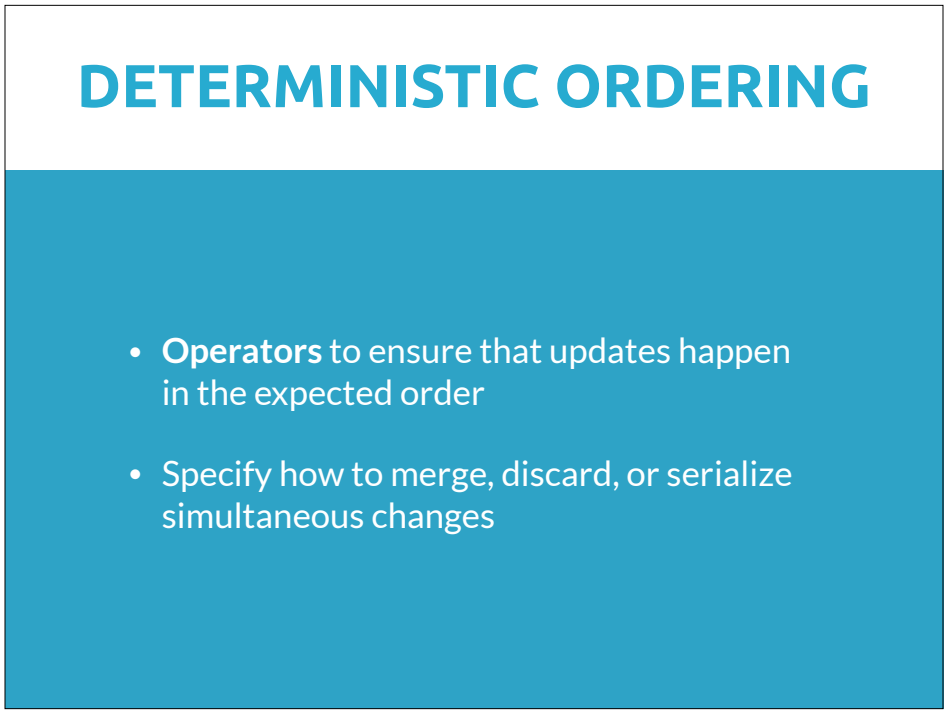
This is an **assignment over time** using ReactiveCocoa. Whenever the text field's content changes, we set `self.name` to it *only if* it's not an empty string.

Deriving state

```
RAC(self, validated) = [RACSignal
  combineLatest:@[
    self.nameField.rac_textSignal,
    self.emailField.rac_textSignal
  ]
  reduce:^(NSString *name, NSString *email)
  {
    return @(name.length > 0 &&
      email.length > 0);
  }];
```

When we treat everything as a stream of values, it becomes incredibly easy to combine and *transform* those streams (just like an immutable object). We can now focus on *deriving* state from input signals—no manual updates are necessary, so it's much harder for data to get out of sync.

This avoids the exponential complexity and invalidation problems associated with mutable state, since we've effectively just designed a pure function.



ReactiveCocoa comes with a large number of operators to manipulate signals. Because *time* is a first-class concept, there are a number of built-in ways to organize values according to the time they arrive.

DETERMINISTIC ORDERING

```
// Pass through updates as they occur
RAC(self, value1) = [RACSignal
  merge:@[ input1, input2 ]];

// Process the input signals in order
RAC(self, value2) = [RACSignal
  concat:@[ input1, input2 ]];

// Combine the values from the inputs
RAC(self, combined) = [RACSignal
  combineLatest:@[ input1, input2 ]];
```

Here are some (non-comprehensive) examples of choosing values based on the time of their occurrence. All of these are deterministic: we get to explicitly specify the order that things should happen in.

This isn't usually possible when mutating a variable in-place. You'd have to use some kind of external ordering mechanism, like a GCD queue (which is a whole 'nother can of worms of its own).

FRP and ReactiveCocoa could easily comprise an entire talk of their own, so I'll leave it at that for now. If you're hungry for more, you can check out reactivecocoa.io for some more philosophy and a link to the repository (which itself contains links to more resources).

We talk about state (and how terrible it is) a *lot* in the ReactiveCocoa community. You can check out issues with the `question` label, or previous Stack Overflow questions, for a peek into some of the philosophy of FRP.

Or just try your hand at pure FP/FRP. Working in a language like Haskell or Elm will open your eyes to how *unnecessary* state really is. Even if you never use them in a real application, they'll teach you valuable lessons that can even be applied to everyday Cocoa programming.

If you want a specific tutorial, I highly recommend Real World Haskell, for a very practical approach to building Real World™ applications in a pure FP language.

Thanks to (in no particular order): Luke Hefson, Josh Abernathy, Josh Vera, Dave Lee, Matt Diephouse, Alan Rogers, Rob Rix, Danny Greg, Robert Böhnke, Andy Matuschak, Jon Sterling, Landon Fuller, Maxwell Swadling, Phillip Bowden, Ash Furrow