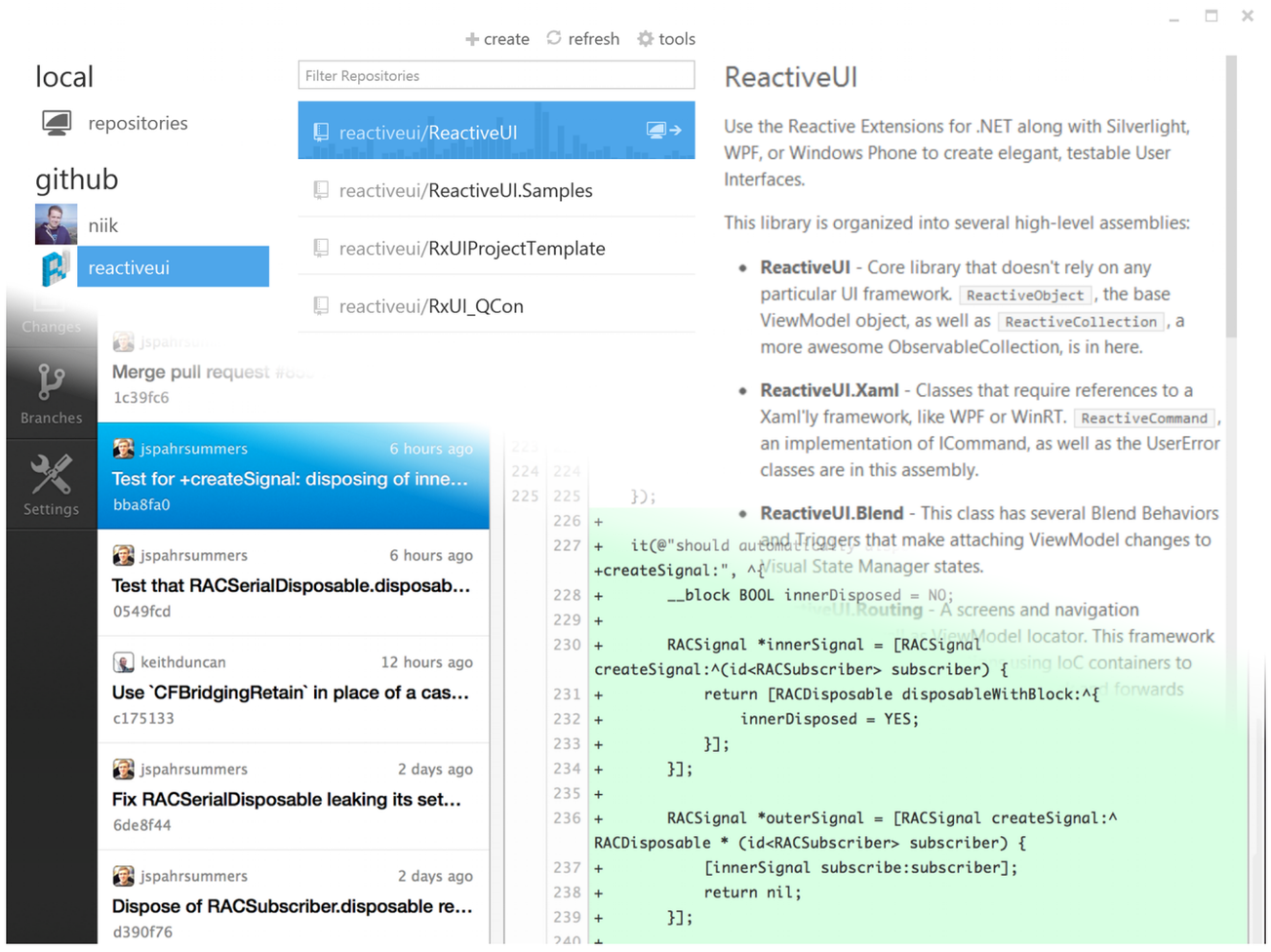
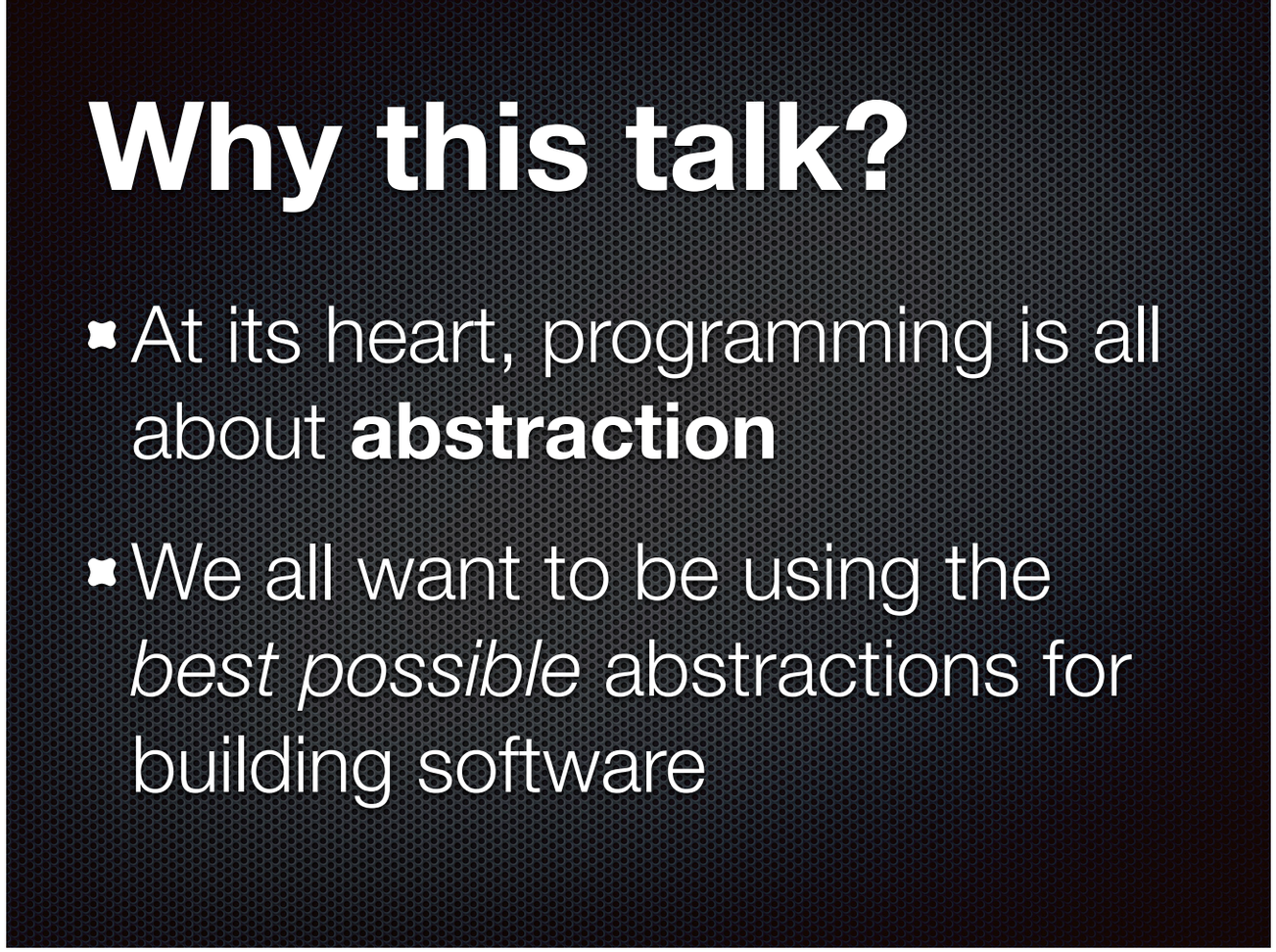
Enemy of the State

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

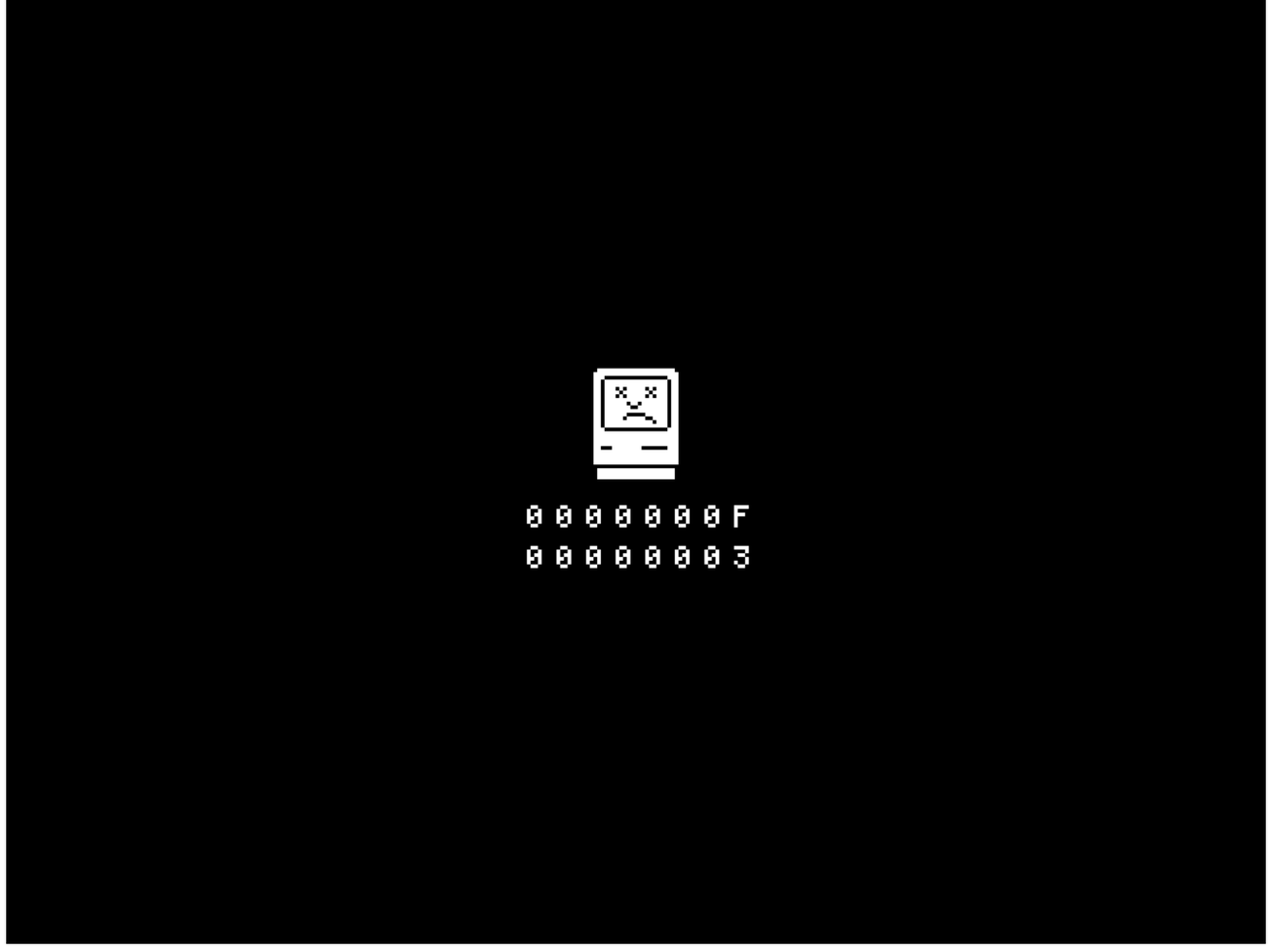
Variables are state

```
int a;

// Store a value
a = 5;

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

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

State is easy

- But **easy** and **simple** are not the same
- A **simple** design minimizes concepts and concerns

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.



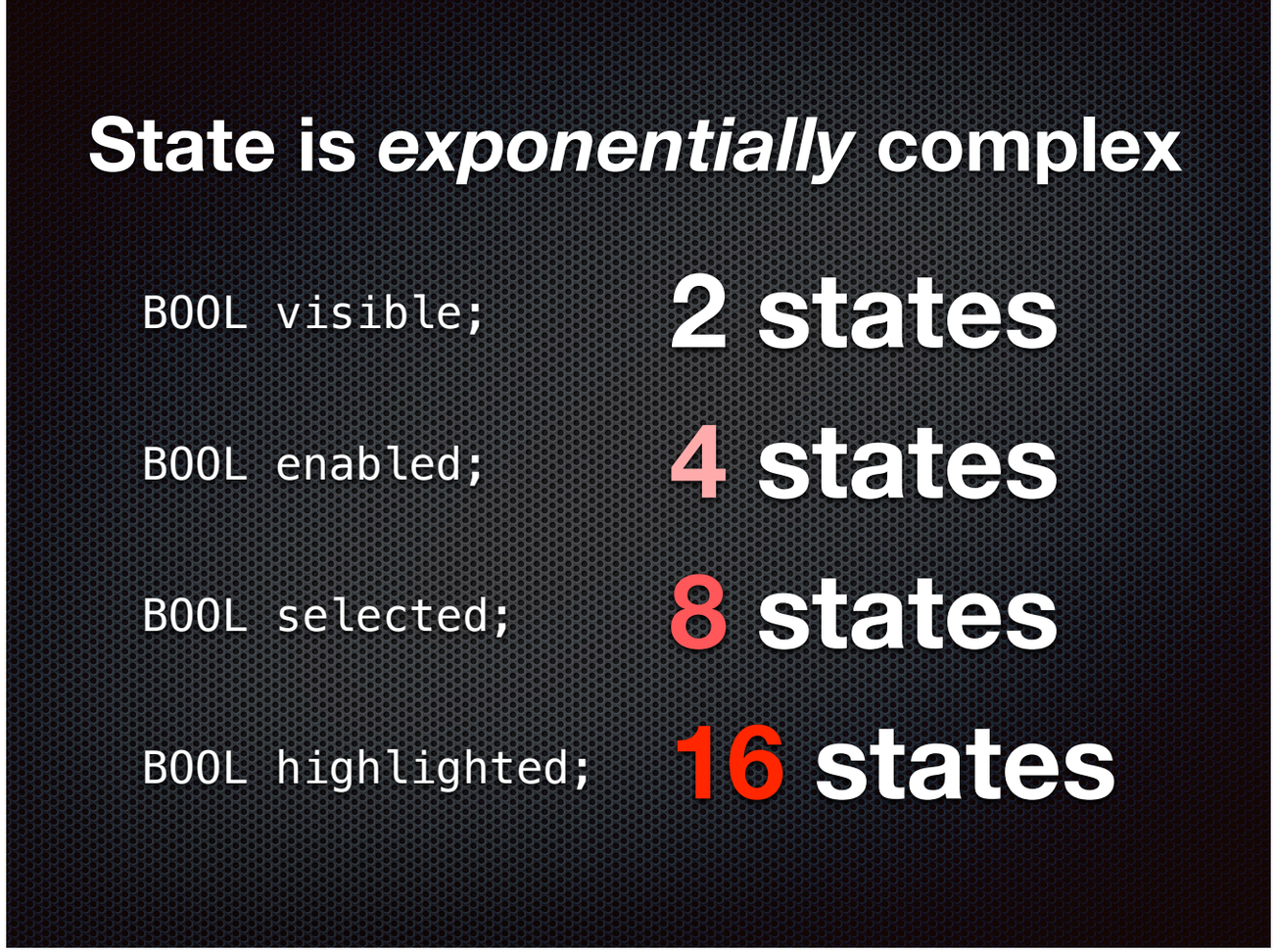
One huge problem with state is that it can "go bad." Any time you've ever restarted a computer or an app, and it fixed an issue you were having, you've been a victim of state.

State is complex

- All systems have some level of **essential** complexity to them
- However, state also adds **incidental** complexity

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

See Rich Hickey's "Simple Made Easy" talk or Moseley and Marks' "Out of the Tar Pit" paper.



As you add each new boolean, you *double* the total number of states that your app can be in. For more complicated data types, the growth in complexity is even more dramatic.

State is a cache

- User interaction often means recalculating (or *invalidating*) some stored state
- Cache invalidation is really, *really* hard to get right

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."

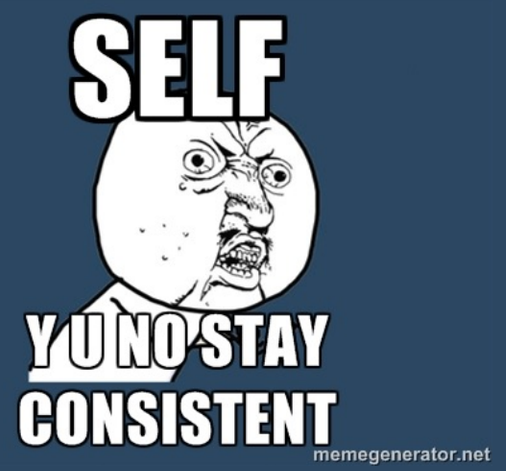
State is nondeterministic

- 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
```



Minimizing state

- Immutability
- Isolation
- Functional reactive programming

Most applications have *some* state, in the form of documents, preferences, caches, the UI, etc. 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.

Immutable objects

- Design classes as **value types**
- Easier to reason about
- No synchronization required for concurrency

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

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
```

Consistency

```
department.employees = newEmployees;

// `department` is inconsistent here!

department.selectedEmployee =
  newEmployees[0];

// Consistency has been restored.
```

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.

Consistency

- Mutable objects can be put in an inconsistent state
- Immutable values only need to be validated at initialization, then never again

Immutable objects can be validated once, and holistically, because all values are set at the same time (initialization). Immutable objects can be designed to make inconsistent/invalid states actually *impossible*.

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;
```

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).

Class clusters

- Transformation is beneficial, but unwieldy and scales poorly
- Class clusters can be used for *temporary* mutability

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.

Minimizing state

- ☑ Immutability
- ☐ Isolation
- ☐ Functional reactive programming

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

Isolation

- The **single responsibility principle** says a class should have one reason to change
- Keep each chunk of state isolated in its own object

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
```

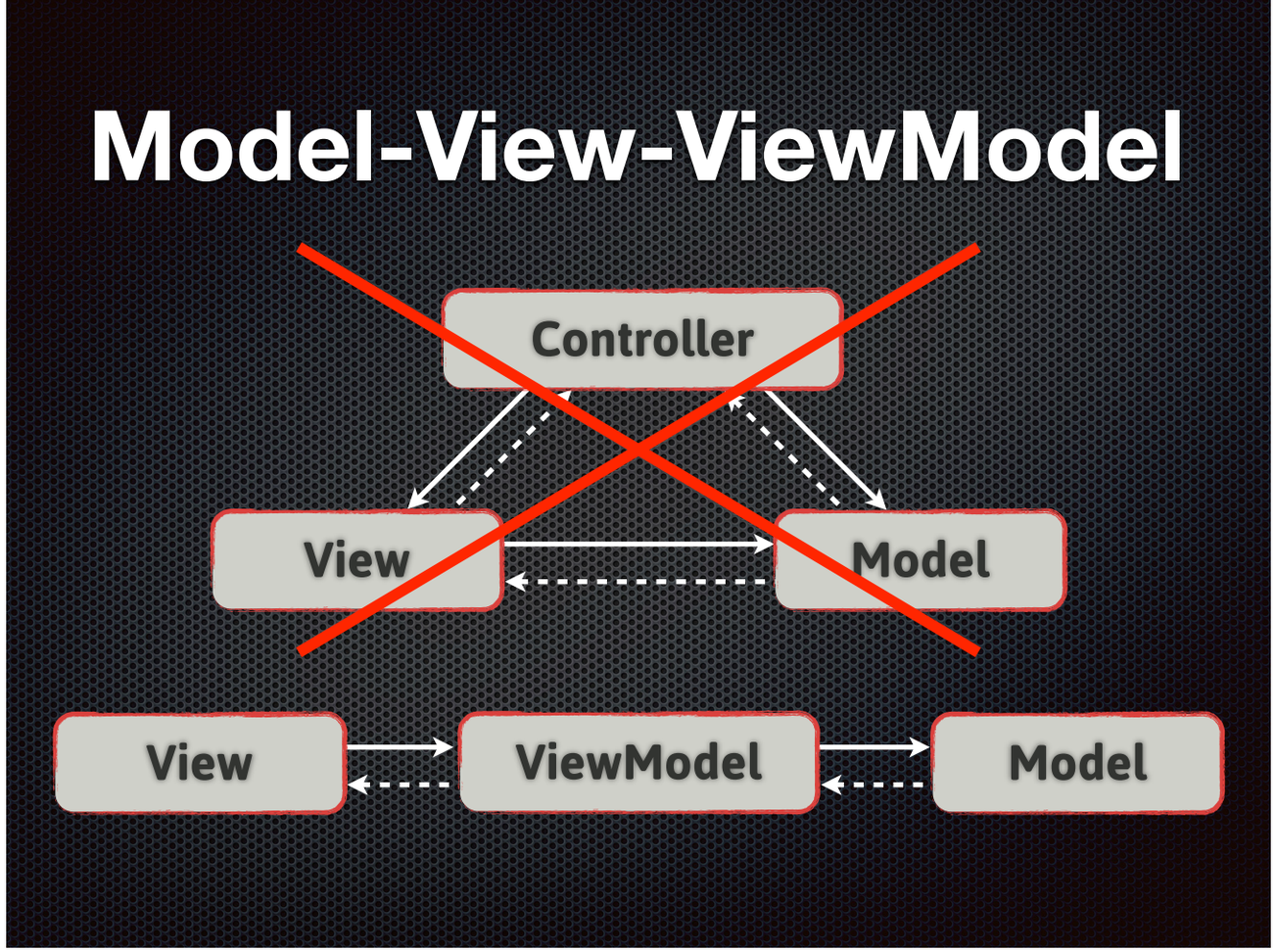
This is an example of poor isolation. The view controller is complecting the concern of *logging in* with the concern of *knowing who's logged in*. As the implementation grows to manage both of these concerns, it becomes difficult to reason about them separately.

Isolation Done *Right™*

```
// For logging in:
@interface LoginViewModel
@property NSString *username;
@property NSString *password;
@end

// After logging in:
@interface UserViewModel
@property User *loggedInUser;
@end
```

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.



MVVM (depicted here, on the bottom), for the uninitiated, 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.

Among many other benefits, this pattern lends itself well to keeping state isolated in the view model. If the view model is responsible for changing the model, you can make the model immutable, and the VM can just apply transformations instead of mutations.

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

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.

Minimizing state

- ☑ Immutability
- ☑ Isolation
- ☐ Functional reactive programming

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

FRP

- Replace variables with series of **values over time**
- Instead of in-place mutation, new values are sent upon a **signal** and can be *reacted* to

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.

Reacting to changes

```
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);
  }];
```

This is an **assignment over time** using ReactiveCocoa, an FRP framework we created. Whenever the name or email text fields change, `self.validated` is updated based on whether both have been filled in.

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

Minimizing state

- ☑ Immutability
- ☑ Isolation
- ☑ Functional reactive programming

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).

Learning more

- Explore discussions around ReactiveCocoa
- Play with purely functional programming languages, like Haskell and Elm

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, and teach you valuable lessons that can 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.

Questions?

Thanks to (in no particular order): Josh Abernathy, Josh Vera, Dave Lee, Matt Diephouse, Alan Rogers, Rob Rix, Robb Böhnke, Andy Matuschak, Jon Sterling