*[Single Responsibility Principle & iOS](http://bendyworks.com/geekville/articles/2014/2/single-responsibility-principle-ios)*

单一职责原则

View Controllers in iOS: we need to talk. You are—without a shadow of a doubt—the worst offender of the Single Responsibility Principle, and that needs to stop.

First, some background

The Single Responsibility Principle or SRP, is defined by Robert Martin (or, more affectionately, "Uncle Bob") in his book[Agile Software Development](http://www.amazon.com/gp/product/0135974445/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0135974445&linkCode=as2&tag=listrophy-20) as the following:

*A class should have only one reason to change.*

This differs from what one might expect the Principle to prescribe. Personally, I had assumed it to mean that a class should have only one reason to act. While this is not a terrible interpretation, it lacks empathy for the software team. By focusing on why the class might *have to be changed*, the SRP allows us to think of how we interact with the code in the present instead of requiring us to project our minds onto the software's future running environment. This, I believe, is an important, and freeing, distinction.

只能有一个原因能让类改变

通俗的说，就是一个类只能有一种职责。

An Example

Let's concoct an example where we need to download a file from our own server. Sure, we could use [AFNetworking](https://github.com/AFNetworking/AFNetworking), but let's assume it doesn't yet exist. A naïve approach might have the following interface:

我们从服务下载一个文件的例子来解释吧。实际环境中，我们肯定会用AFNetworking，但我们假定它并不存在。代码如下：

@interface Downloader - (void)download:(NSString \*)path withSuccess:(void (^)(id resp))success; @end @interface Downloader () - (void)buildRequestObject; - (void)start; - (void)dataChunkWasReceived:(NSData \*)data; - (void)downloadDidFinish; - (void)closeSession; @end

Why might we ever want to change this class? I can think of two reasons immediately. First, our operations team may have added SSL/TLS, and we must use a custom certificate. Second, our server application team may add JSON as a MIME type and intend to deprecate SOAP (*fistpump*). In this situation, the Single Responsibility Principle advises us to turn our Downloader class into

a [Facade](http://www.dofactory.com/Patterns/PatternFacade.aspx) for DownloaderConnection and DownloaderParser classes... or class extensions if you wish.

@interface DownloaderConnection () // or @interface Downloader (Connection) - (void)buildRequestObject; - (void)start; - (void)closeSession; @end @interface DownloaderParser () // or @interface Downloader (Parser) - (void)dataChunkWasReceived:(NSData \*)data; - (void)downloadDidFinish; @end

Now we know where to attack a changing business requirement such as "let's use an internal server if we're on the company WiFi."

The 900-Line Problem

If you've had any exposure to sample code, template code, or production code for iOS, your brain has already made the connection to how View Controller usage usually ignores SRP completely. For those without such exposure and for those of us who like counting our scars, let's enumerate some reasons why we might need to change a View Controller:

* The API to our datastore changed.
* A UIActionSheet needs to be converted to a full-blown popover or presentation view.
* An animation needs tweaking.
* A new business requirement where we need to capture a phone number of a contact in anABPeoplePickerNavigationController instead of just an email.
* The "save" button needs stricter validation.
* Tapping a particular button needs to trigger a different selector.
* The position of a subview requires a complex calculation that has many edge cases, many of which are not yet discovered.
* And myriad other reasons relating to the domain of the application.

The end result of all of these loci of change can be characterized as a 900-line file of application and business logic in a form much resembling spaghetti. And perhaps "900 lines" is too charitable.

The only type of file that may have 900 lines is one that is never intended for human consumption—binary files and database dumps being the prime examples. Using #pragma mark - is indicative of a failure to adhere to the Single Responsibility Principle and is a signpost for poor code.

A Heuristic Approach

Faced with such a daunting task, we as a community may throw—and indeed have thrown—in the towel. "It's just code," we might say. "As long as it works, the architecture shouldn't matter that much." This is where you might be unsurprised to hear me disagree. Code is written for other humans, otherwise we'd still be writing microcode. Architecture matters, and the accepted behavior of "just throw it in the VC" is harmful to everyone from novices to high-functioning teams of experts.

So, we want to reduce the number of reasons why we might change a View Controller to one. In order to do that, let us just pick one of the reasons and exclude all others:

*View Controllers may only contain code that effects view changes in response to data changes.*

One might think, "I already write my View Controllers in that way!" Well, to be sure we're on the same page, allow me propose rules that must be followed in order to adhere to this heuristic:

* The IBAction macro must not be used in a View Controller
* The @interface block in a View Controller's header file must be blank.
* A View Controller may not implement any extra \*Delegate or \*DataSource protocols except for observing Model changes.
* A View Controller may only do work in viewDidLoad (or awakeFromNib), viewDidAppear, viewDidDisappear, and in response to an observed change in the model layer.

To abide by these rules is to respect the Single Responsibility Principle in View Controllers.

How Do You Propose We Do THAT?!

First off, Apple helpfully provides a number of mechanisms to do just that. For example, UITableViewDataSourceallows us to convert a database into an enumeration of domain objects, and UIAlertViewDelegate provides a mechanism to handle a user's choice on a UIAlertView. To follow the SRP with these protocols is to apply them to objects **other** than View Controllers.

For more custom behavior, let's define a new class of objects that turn view interactions into model updates. I haven't found a name for such an object class in the literature, perhaps other than Controller, so let's also invent a name for it: Intention.

To use an Intention without polluting a View Controller, we embed it in a Storyboard Scene or NIB. Allowing for many Intention objects per scene, we can use them to interact directly with your Model layer, be that a mutable data structure in an IBOutlet or CoreData or Couchbase or whatever. An Intention does not know about the View Controller and vice versa.

By architecting our applications in this way, we more fully embrace the Model-View-Controller (MVC) pattern that Cocoa allegedly promotes. We decouple interactions from intents, intents from results, and results from affordances.

Intentions in Action

Let's implement a simple email field with a button that enables/disables based on a valid email address using Core Data. First, let's define our intentions:

* EnterEmailIntention: Updating the email field should toggle the save button depending on validation.
* SaveEmailIntention: Tapping the button should save the NSManagedObjectContext (MOC).

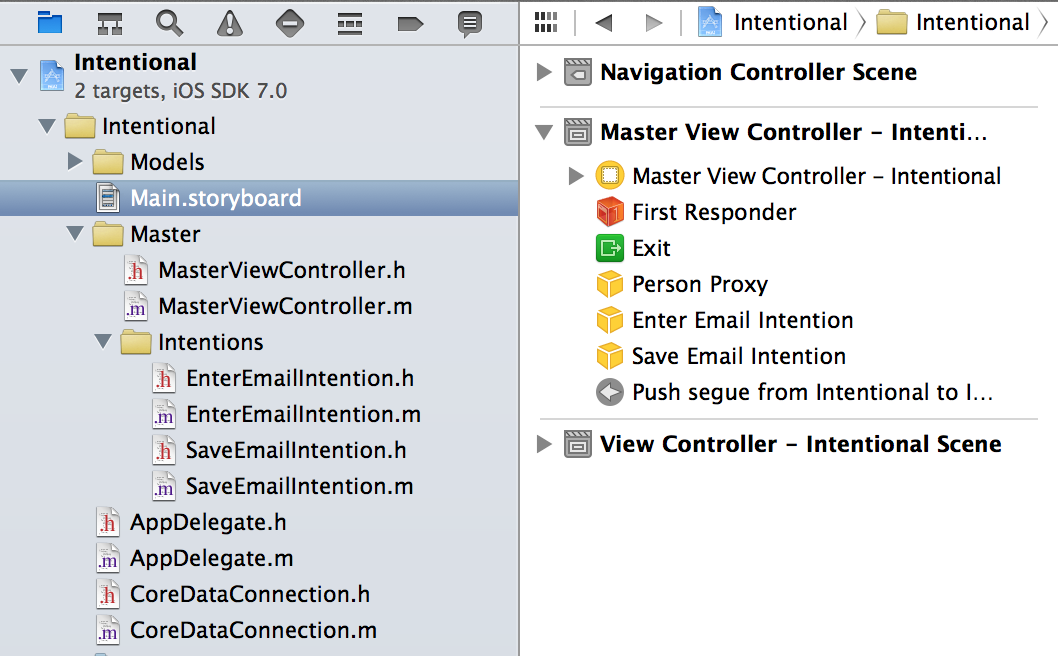
Our data model is simple enough: A Person entity with an email attribute with a regex validation of[^@]+@(?:[^@\.]+\.)+[a-z]{2,6} (NB: don't use this regex in production!).

By constructing the appropriate classes for these Intentions and creating a PersonProxy for Person, we end up with the following implementation for our View Controller:

@implementation MasterViewController - (void)viewDidLoad { [super viewDidLoad]; NSNotificationCenter \*center = [NSNotificationCenter defaultCenter]; [center addObserver:self selector:@selector(emailChanged:) name:NSManagedObjectContextObjectsDidChangeNotification object:self.managedObjectContext]; [center addObserver:self selector:@selector(personSaved:) name:NSManagedObjectContextDidSaveNotification object:self.managedObjectContext]; [self setEnabledForSaveButton]; } - (void)dealloc { [[NSNotificationCenter defaultCenter] removeObserver:self]; } // WARNING: A big assumption was made that any change to the MOC // must be because the email textField changed. Typically, you'd switch on the // notification's NSUpdatedObjectsKey in the userInfo. - (void)emailChanged:(NSNotification \*)note { [self setEnabledForSaveButton]; } - (void)setEnabledForSaveButton { [self.saveButton setEnabled:[self.personProxy.person validateForInsert:nil]]; } - (void)personSaved:(NSNotification \*)note { [self performSegueWithIdentifier:@"advance" sender:note]; } @end

That's it! As long as we implement our Intention classes and connect the IBOutlet variables correctly in the Storyboard, our View Controller now only implements what we sought: only effect view changes in response to data changes. Not only that, but our Intention classes obey SRP as well, since we know exactly which class to approach given new requirements.

The file and scene layout is seen here:



A [sample app](https://github.com/listrophy/Intentional) is on GitHub.

How Does It Feel?

After implementing this separation of concerns the first time, I felt free. "This," I thought, "feels **clean**. I *like* this." As I continued with my work, I added another View Controller to my project. Expecting it to be one of those "simple" VCs, I thought I could get away with not using this new scheme. How wrong I was. I quickly gave in to this new architecture and immediately felt better about the code.

Extracting all interaction and storage logic from a View Controller leaves a lean, manageable class. If you're familiar with the term, we've basically turned our View Controller into a View Model.