Think Small

Big Improvements Can Come in
Tiny Classes

Greetings…

I want you to understand how I think about creating small generic solutions.

It’s not the code I’m presenting… It’s the thought process.

The work is stripping everything away until you only have the tiny core left. What do I mean by Tiny Core?

Common Higher-Order Functions

- Map
  - Map an Array to a New Array With a Unary Operation
- Filter
  - Filter an Array With a Unary Predicate
- Reduce (Fold/Accumulate/Aggregate/Compress/Inject)
  - Reduce an Array to an Object With a Binary Operation

I think an example this idea are higher order functions.

Who here has not heard of map, filter, and reduce?

Map, filter, and reduce have proven to be invaluable tools in many languages, for some reason missing in the Foundation Framework.

They are simple and flexible and they can be implemented in only a few lines of code. What does this look like in Objective C?

```objc
#import <Foundation/Foundation.h>

@interface NSArray (JLTMapFilterReduce)

- (NSArray *)arrayByMapping:(id(^)(id obj))mapping;

- (NSArray *)arrayByFiltering:(BOOL(^)(id obj))filtering;

- (id)objectByReducing:(id(^)(id obj1, id obj2))reducing;

- (id)objectByReducing:(id(^)(id obj1, id obj2))reducing
         initialObject:(id)initialObject;

- (NSArray *)arrayByRemovingFirstObject;

@end
```

Here is a tiny category for NSArray: as promised mapping, filtering, and reducing.

This follows the common naming system in Objective C: the name is the return type and a prepositional phrase to describe the method.

I provide two version of reduce, the second one takes a seed value.

I also threw in cdr/rest/tail as -arrayByRemovingFirstObject.

```objc
#import "NSArray+JLTMapFilterReduce.h"

@implementation NSArray (JLTMapFilterReduce)

- (NSArray *)arrayByMapping:(id (^)(id))mapping
{
    NSMutableArray *result = [NSMutableArray arrayWithCapacity:[self count]];
    for (id obj in self)
        [result addObject:mapping(obj)];
    return [result copy];
}

- (NSArray *)arrayByFiltering:(BOOL (^)(id))filtering
{
    NSMutableArray *result = [NSMutableArray arrayWithCapacity:[self count]];
    for (id obj in self)
        if (filtering(obj))
            [result addObject:obj];
    return [result copy];
}

- (id)objectByReducing:(id(^)(id, id))reducing
{
    return [[self arrayByRemovingFirstObject] objectByReducing:reducing initialObject:[self firstObject]];
}

- (id)objectByReducing:(id (^)(id, id))reducing initialObject:(id)initialObject
{
    id result = initialObject;
    for (id obj in self)
        result = reducing(result, obj);
    return result;
}

- (NSArray *)arrayByRemovingFirstObject
{
    return [self count] > 1 ? [self subarrayWithRange:NSMakeRange(1, [self count] - 1)] : @[];
}

@end
```

The implementation is straight forward.

As you can see, all of this can be done in just a few lines of code.

You can also see why I threw in -arrayByRemovingFirstObject.

Tiny Solutions

- Extract Often Repeated Logic into Classes/Categories
- Simple Generic Solutions Do the Work Right Now
- Simple Generic Solutions Are Flexible for the Future

Here is the focus of my talk. Often repeated snippets of code can be extracted into tiny classes and categories.

Sometimes it’s extra effort to extract the tiny core from a chunk of often repeated code.

The effort is worth the reward. What is the reward you ask? The reward is time saved in the future.

I don’t like alert views.

The delegation pull you out of the current context and forces you to reconstruct that context in the delegate method.

```objc
- (IBAction)buttonTextToMagic:(UIButton *)button
{
    NSString *message = @"Make the button title \"Magic!\"?";
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Make Magic"
                                                        message:message
                                                       delegate:self
                                              cancelButtonTitle:@"NO"
                                              otherButtonTitles:@"YES", nil];
    self.jlt_buttonForAlertView = button;
    [alertView show];
}

- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
    buttonIndex
{
    if (self.jlt_buttonForAlertView != nil) {
        if (buttonIndex != alertView.cancelButtonIndex) {
            UIButton *button = self.jlt_buttonForAlertView;
            [button setTitle:@"Magic!" forState:UIControlStateNormal];
        }
        self.jlt_buttonForAlertView = nil;
    }
}
```

You all have seen this code before.

I used an extra property in the view controller just to hold the context.

What's Wrong With That

- The View Controller Needs to Keep Data Only the Alert View Cares About
- The Data for the Alert View Has a Different Lifecycle Than the Alert View
- Care Is Needed to Keep Data Distinct

The view controller has data which doesn't belong to it.

JLTContextInfo Goals

- Keep Context Specific Data From Polluting the View Controller
- Give Alert View Specific Data the Same Lifecycle as the Alert View
- Distinguish the Alert View Specific Data

This can be done with a mutable dictionary attached to every NSObject.

```objc
@interface NSObject (JLTContextInfo)

@property (nonatomic, readonly) NSMutableDictionary *contextInfo;

@end

@implementation NSObject (JLTContextInfo)

- (NSMutableDictionary *)contextInfo
{
    static char key;
    NSMutableDictionary *instance = objc_getAssociatedObject(self, &key);

    if (!instance) {
        instance = [NSMutableDictionary dictionary];
        objc_setAssociatedObject(self, &key, instance,
                                 OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    }

    return instance;
}

@end
```

Exactly as promised.

The core idea is extending the data available to a class in a small but useful way.

```objc
- (IBAction)buttonTextToMagic:(UIButton *)button
{
    NSString *message = @"Make the button title \"Magic!\"?";
    UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Make Magic"
                                                        message:message
                                                       delegate:self
                                              cancelButtonTitle:@"NO"
                                              otherButtonTitles:@"YES", nil];
    alertView.contextInfo[@"button"] = button;
    [alertView show];
}

- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)
    buttonIndex
{
    if (alertView.contextInfo[@"button"] != nil) {
        if (buttonIndex != alertView.cancelButtonIndex) {
            UIButton *button = alertView.contextInfo[@"button"];
            [button setTitle:@"Magic!" forState:UIControlStateNormal];
        }
    }
}
```

The code is looks nearly identical… but the meaning behind it is so much more.

- The alert view is combined with it’s context.
- The view controller’s namespace doesn’t get polluted with irrelevant properties.
- The lifecycle of the alert view and it’s context data are automatically kept in sync.

So you would think this made me happy… but no.

Fail!
There's a Better Solution

- iOS Programming Recipe 22:
  Simplify UIAlertView With Blocks
  - http://nscookbook.com/2013/04/ios-programming-recipe-22-simplify-uialertview-with-blocks/

Well… OK… that was a bust.

But wait, is that all… none of contextInfo is UIAlertView specific.

Remember Simple Generic Solutions Are Flexible

Now that I got the code, what else can I do with it.

```objc
- (CGFloat)tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    JLTDemoModel *datum = self.data[indexPath.row];

    return [datum.contextInfo[@"isExpanded"] boolValue] ? 88.0 : 44.0;
}

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    JLTDemoModel *datum = self.data[indexPath.row];

    BOOL isExpanded = [datum.contextInfo[@"isExpanded"] boolValue];
    datum.contextInfo[@"isExpanded"] = !isExpanded ? @YES : @NO;

    [tableView beginUpdates];
    [tableView endUpdates];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Here is a sample that uses contextInfo to attach cell metadata onto the underlying data source.

This prevented me from keeping parallel arrays or looking up the metadata in a dictionary.

```
@interface JLTDemoModel (JLTExpandable)
@property (nonatomic, getter = isExpanded) BOOL expanded;
@end

@implementation JLTDemoModel (JLTExpandable)
- (BOOL)isExpanded
{
    return [self.contextInfo[@"isExpanded"] boolValue];
}
- (void)setExpanded:(BOOL)expanded
{
    if (expanded)
        self.contextInfo[@"isExpanded"] = @YES;
    else
        [self.contextInfo removeObjectForKey:@"isExpanded"];
}
@end

@implementation JLTTableViewController

- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath
{
    JLTDemoModel *datum = self.data[indexPath.row];

    return datum.expanded ? 88.0 : 44.0;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    JLTDemoModel *datum = self.data[indexPath.row];

    datum.expanded = !datum.expanded;

    [tableView beginUpdates];
    [tableView endUpdates];
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Here’s a example category with properties. Because contextInfo does all the associated object work itself, backing properties with data looks cleaner and reduces cognitive complexity.

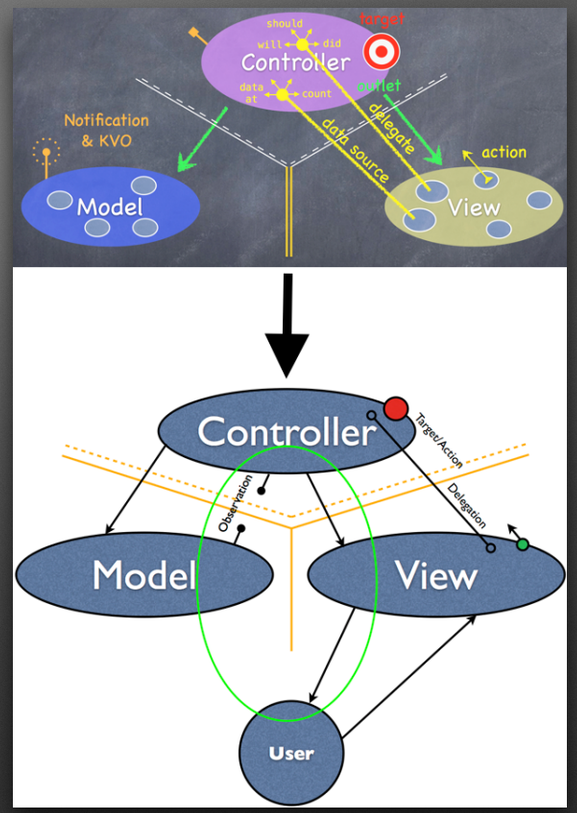
```objc
- (IBAction)connectToGoogle:(UIButton *)button
{
    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:@"http://google.com/"]];
    NSURLConnection *connection = [NSURLConnection connectionWithRequest:request delegate:self];
    connection.contextInfo[@"button"] = button;
}

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response
{
    connection.contextInfo[@"response"] = response;
    connection.contextInfo[@"responseData"] = [NSMutableData data];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [connection.contextInfo[@"responseData"] appendData:data];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSHTTPURLResponse *response = connection.contextInfo[@"response"];
    NSData *responseData = connection.contextInfo[@"responseData"];
    NSInteger statusCode = [response statusCode];
    NSInteger dataLength = [responseData length];

    if (connection.contextInfo[@"button"] != nil) {
        UIButton *button = connection.contextInfo[@"button"];
        NSString *title = [NSString stringWithFormat:@"%ld-%ld", (long)statusCode, (long)dataLength];
        [button setTitle:title forState:UIControlStateNormal];
    }
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    NSLog(@"Error: %@", error);

    if (connection.contextInfo[@"button"] != nil) {
        UIButton *button = connection.contextInfo[@"button"];
        NSString *title = @"Connection Failed";
        [button setTitle:title forState:UIControlStateNormal];
    }
}
```

Finally, the real power. NSURLConnection which is re-entrant.

This object can make parallel requests and they don’t interfere with each other.

MVC

The Problem of Concise Order



Let’s leave the problem of context behind and go onto Apple’s beautiful and terrible MVC.

Upper MVC Graphic is from Developing iOS7 Apps for iPad and iPhone I Lecture 1 Slides I Page 37

```objc
- (IBAction)loginUser:(id)sender
{
    [self jlt_performAuthentication];
    self.password = nil;
    self.passwordTextField.text = nil;
    self.loginButton.enabled = NO;
}

- (IBAction)resetForm:(id)sender
{
    self.username = nil;
    self.password = nil;
    self.usernameTextField.text = nil;
    self.passwordTextField.text = nil;
    self.loginButton.enabled = NO;
    self.resetButton.enabled = NO;
}

- (BOOL)textField:(UITextField *)textField shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString *)string
{
    NSString *text = [textField.text stringByReplacingCharactersInRange:range withString:string];

    if (textField == self.usernameTextField) {
        self.username = text;
    } else if (textField == self.passwordTextField) {
        self.password = text;
    }

    self.loginButton.enabled = [self.username length] > 0 && [self.password length] > 0;
    self.resetButton.enabled = [self.username length] > 0 || [self.password length] > 0;

    return YES;
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    self.usernameTextField.text = self.username;
    self.passwordTextField.text = self.password;
    self.loginButton.enabled = [self.username length] > 0 && [self.password length] > 0;
    self.resetButton.enabled = [self.username length] > 0 || [self.password length] > 0;
}
```

I've seen a lot of this code… it's alway made me feel nauseated.

The UI elements are initialized as a separate step.

Each action must manage all the needed changes of the UI elements.

What's Wrong With That

- Initial View Updates Are Done Differently Than Action and Delegate View Updates
- Each Action or Delegate Needs to Ensure Every Effected View Is Updated Correctly
- Model and View Changes Are Combined

It's hard to pick out model changes with all the mixed in UI code.

There is common UI logic which is repeated multiple times.

Subjectively, it looks confusing and messy.

My Way to MVC

- Actions and Delegates Only Update the Model
- KVO and Notifications Update Views

```objc
- (IBAction)loginUser:(id)sender
{
    [self jlt_performAuthentication];
    self.password = nil;
}

- (IBAction)resetForm:(id)sender
{
    self.username = nil;
    self.password = nil;
}

- (BOOL)jlt_canLogin
{
    return [self.username length] > 0 && [self.password length] > 0;
}

- (BOOL)jlt_canReset
{
    return [self.username length] > 0 || [self.password length] > 0;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    if ([keyPath isEqualToString:@"username"]) {
        if (!self.usernameTextField.editing)
            self.usernameTextField.text = self.username;
    } else if ([keyPath isEqualToString:@"password"]) {
        if (!self.passwordTextField.editing)
            self.passwordTextField.text = self.password;
    } else if ([keyPath isEqualToString:@"jlt_canLogin"]) {
        self.loginButton.enabled = self.jlt_canLogin;
    } else if ([keyPath isEqualToString:@"jlt_canReset"]) {
        self.resetButton.enabled = self.jlt_canReset;
    }
}
```

I have a whole talk on just this one subject.

The TL;DR of it is actions update the model/KVO updates the UI.

```objc
- (IBAction)loginUser:(id)sender
{
    [self jlt_performAuthentication];
    self.password = nil;
}

- (IBAction)resetForm:(id)sender
{
    self.username = nil;
    self.password = nil;
}

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
                        change:(NSDictionary *)change context:(void *)context
{
    if ([keyPath isEqualToString:@"username"]) {
        if (!self.usernameTextField.editing)
            self.usernameTextField.text = self.username;
    } else if ([keyPath isEqualToString:@"password"]) {
        if (!self.passwordTextField.editing)
            self.passwordTextField.text = self.password;
    } else if ([keyPath isEqualToString:@"jlt_canLogin"]) {
        self.loginButton.enabled = self.jlt_canLogin;
    } else if ([keyPath isEqualToString:@"jlt_canReset"]) {
        self.resetButton.enabled = self.jlt_canReset;
    }
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self addObserver:self forKeyPath:@"username" options:NSKeyValueObservingOptionInitial context:NULL];
    [self addObserver:self forKeyPath:@"password" options:NSKeyValueObservingOptionInitial context:NULL];
    [self addObserver:self forKeyPath:@"jlt_canLogin" options:NSKeyValueObservingOptionInitial context:NULL];
    [self addObserver:self forKeyPath:@"jlt_canReset" options:NSKeyValueObservingOptionInitial context:NULL];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [self removeObserver:self forKeyPath:@"username"];
    [self removeObserver:self forKeyPath:@"password"];
    [self removeObserver:self forKeyPath:@"jlt_canLogin"];
    [self removeObserver:self forKeyPath:@"jlt_canReset"];
    [super viewWillDisappear:animated];
}
```

There are some things missing here:

Using the UITextFieldDelegate to set the username and password.

jlt_canLogin and jlt_canReset use +keyPathsForValuesAffecting<Key>.

What's Wrong With That

- Separate Logic for Beginning and Ending the Observing
- Lots of Boilerplate Code
- Can't Distinguish Between Initial View Update and Model Change Updates

It all boils down to -viewWillAppear: and -viewWillDisappear: get ugly.

JLTEasyObservation Goals

- Begin and End All Observers and Notifications at the Same Time
- Remove the Boilerplate Code
- Provide the Observer With a Way to Distinguish Between Initial View Update and Model Change Updates

So, fix the pain point in -viewWillAppear: and -viewWillDisappear:.

```
@interface UIViewController (JLTEasyObservation)

- (void)beginEasyObserving;
- (void)endEasyObserving;

@property (copy, nonatomic) NSArray *easyObservationKeyPaths;
@property (copy, nonatomic) NSArray *easyNotificationNames;

@property (nonatomic, readonly,
           getter = isEasyObserving) BOOL easyObserving;

@property (nonatomic, readonly,
           getter = isInitialEasyObservation) BOOL initialEasyObservation;

@end

@protocol JLTEasyNotifying <NSObject>

- (void)observeEasyNotification:(NSNotification *)notification;

@end
```

Use an array to hold key paths to observe.

Provide methods to begin and end observing all those keys as once.

Set a flag to determine the initial KVO.

I even threw in a system for notifications to boot.

```objc
@implementation UIViewController (JLTEasyObservation)

- (void)beginEasyObserving
{
    if (self.easyObserving) return;

    self.easyObserving = YES;

    SEL selector = @selector(observeEasyNotification:);
    if ([self respondsToSelector:selector])
        for (NSString *name in self.easyNotificationNames)
            [[NSNotificationCenter defaultCenter] addObserver:self selector:selector name:name object:nil];

    self.initialEasyObservation = YES;
    for (NSString *keyPath in self.easyObservationKeyPaths)
        [self addObserver:self forKeyPath:keyPath options:NSKeyValueObservingOptionInitial context:NULL];
    self.initialEasyObservation = NO;
}

- (void)endEasyObserving
{
    if (!self.easyObserving) return;

    SEL selector = @selector(observeEasyNotification:);
    if ([self respondsToSelector:selector])
        for (NSString *name in self.easyNotificationNames)
            [[NSNotificationCenter defaultCenter] removeObserver:self name:name object:nil];

    for (NSString *keyPath in self.easyObservationKeyPaths)
        [self removeObserver:self forKeyPath:keyPath context:NULL];

    self.easyObserving = NO;
}

- (BOOL)isEasyObserving {...}
- (NSArray *)easyObservationKeyPaths {...}
- (void)setEasyObservationKeyPaths:(NSArray *)easyObservationKeyPaths {...}
- (NSArray *)easyNotificationNames {...}
- (void)setEasyNotificationNames:(NSArray *)easyNotificationNames {...}
- (BOOL)isInitialEasyObservation {...}
- (void)setEasyObserving:(BOOL)easyObserving {...}
- (void)setInitialEasyObservation:(BOOL)initialEasyObservation {...}
- (NSMutableDictionary *)JLT_easyObservationDict {...}

@end
```

The core idea is a small extension to a class to handle boiler plate code.

```objc
@implementation UIViewController (JLTEasyObservation)

- (void)beginEasyObserving {...}
- (void)endEasyObserving {...}

- (BOOL)isEasyObserving {
    return [self.JLT_easyObservationDict[@"easyObserving"] boolValue];
}
- (NSArray *)easyObservationKeyPaths {
    return self.JLT_easyObservationDict[@"easyObservationKeyPaths"];
}
- (void)setEasyObservationKeyPaths:(NSArray *)easyObservationKeyPaths {
    BOOL observing = self.easyObserving;

    if (observing) [self endEasyObserving];
    self.JLT_easyObservationDict[@"easyObservationKeyPaths"] = [easyObservationKeyPaths copy];
    if (observing) [self beginEasyObserving];
}
- (NSArray *)easyNotificationNames {
    return self.JLT_easyObservationDict[@"easyNotificationNames"];
}
- (void)setEasyNotificationNames:(NSArray *)easyNotificationNames {
    BOOL observing = self.easyObserving;

    if (observing) [self endEasyObserving];
    self.JLT_easyObservationDict[@"easyNotificationNames"] = [easyNotificationNames copy];
    if (observing) [self beginEasyObserving];
}
- (BOOL)isInitialEasyObservation {
    return [self.JLT_easyObservationDict[@"initialEasyObservation"] boolValue];
}
- (void)setEasyObserving:(BOOL)easyObserving {
    self.JLT_easyObservationDict[@"easyObserving"] = easyObserving ? @YES : @NO;
}
- (void)setInitialEasyObservation:(BOOL)initialEasyObservation {
    self.JLT_easyObservationDict[@"initialEasyObservation"] = initialEasyObservation ? @YES : @NO;
}
- (NSMutableDictionary *)JLT_easyObservationDict {
    static char key;
    NSMutableDictionary *dict = objc_getAssociatedObject(self, &key);
    if (!dict) {
        dict = [NSMutableDictionary dictionary];
        objc_setAssociatedObject(self, &key, dict, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    }
    return dict;
}

@end
```

For all of the properties, I reused the context info idea.

```objc
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.easyObservationKeyPaths = @[@"username", @"password", @"jlt_canLogin", @"jlt_canReset"];
    self.easyNotificationNames = @[UIKeyboardWillChangeFrameNotification, UIKeyboardWillHideNotification];
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self beginEasyObserving];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [self endEasyObserving];
    [super viewWillDisappear:animated];
}

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
                        change:(NSDictionary *)change context:(void *)context
{
    if ([keyPath isEqualToString:@"username"]) {
        if (!self.usernameTextField.editing)
            self.usernameTextField.text = self.username;
    } else if ([keyPath isEqualToString:@"password"]) {
        if (!self.passwordTextField.editing)
            self.passwordTextField.text = self.password;
    } else if ([keyPath isEqualToString:@"jlt_canLogin"]) {
        self.loginButton.enabled = self.jlt_canLogin;
    } else if ([keyPath isEqualToString:@"jlt_canReset"]) {
        self.resetButton.enabled = self.jlt_canReset;
    }
}

- (void)observeEasyNotification:(NSNotification *)notification
{
    if ([notification.name isEqualToString:UIKeyboardWillChangeFrameNotification] ||
        [notification.name isEqualToString:UIKeyboardWillHideNotification]) {
        UIViewAnimationCurve curve = [notification.userInfo[UIKeyboardAnimationCurveUserInfoKey] intValue];
        NSTimeInterval duration = [notification.userInfo[UIKeyboardAnimationDurationUserInfoKey] floatValue];
        CGRect frameEnd = [notification.userInfo[UIKeyboardFrameEndUserInfoKey] CGRectValue];

        frameEnd = [self.view convertRect:frameEnd fromView:nil];
        CGFloat offset = self.view.bounds.size.height - frameEnd.origin.y;

        [self.view layoutIfNeeded];
        [UIView animateWithDuration:duration delay:0.0 options:curve << 16 animations:^{
            self.jlt_keyboardMarkerBottomConstraint.constant = offset;
            [self.view layoutIfNeeded];
        } completion:NULL];
    }
}
```

This example got so short, so I threw in a notification piece onto it.

By the way, I'm still thinking about cleaning up keyboard events.

If you have any ideas cleaning that up, let me know.

Works With Existing UIViewControllers Subclasses

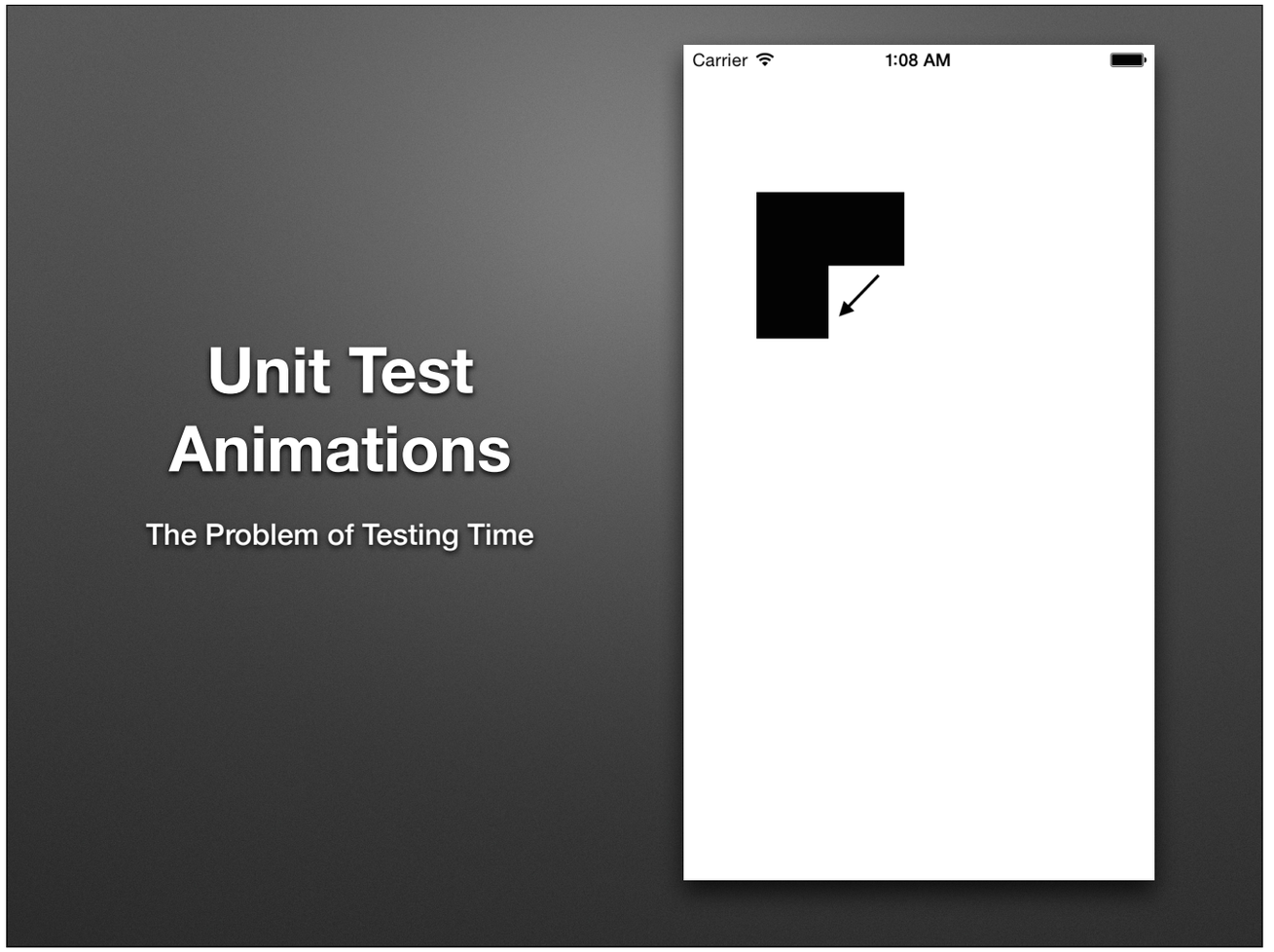
- It's a Category, Not a Subclass: so No Need to Change the View Controller's Interface
- Methods Which Don't Conform Still Work and Don't Interfere With Easy Observing
- It's Possible to Convert a Whole Controller by Converting Only One Method at a Time

The sales pitch…

All of this can be yours for free!

Don't wait; act now.

Github is standing by.



Moving on from MVC, I now will look at time.

Specifically, how to unit test things which take time.

I ran into this problem while building a custom view with animations.

```
@implementation JLTViewController

- (IBAction)swapHeightAndWidth:(id)sender
{
    [self swapHeightAndWidthWithDuration:2.0 completion:NULL];
}

- (void)swapHeightAndWidthWithDuration:(NSTimeInterval)duration
                            completion:(void (^)())completion
{
    CGRect frame = self.rectangleView.frame;

    frame.size.height = self.rectangleView.frame.size.width;
    frame.size.width = self.rectangleView.frame.size.height;

    [UIView animateWithDuration:duration animations:^{
        self.rectangleView.frame = frame;
    } completion:^(BOOL finished) {
        if (completion) completion();
    }];
}

@end
```

Here we have a simple animation, which no-one would ever want to do.

What's Wrong With That

- Difficult to Unit Test

What else can I say… unit testing an animation sucks.

JLTWaiter Goals

- Pause Execution of a Unit Test to Wait for an Async Task
- Continue Execution When the Async Task Completes or After a Specified Timeout
- Determine If the Async Task Completed

This seem like it would be easy… no wait, I mean very difficult.

Luckily, Objective C provides -[NSRunLoop runUntilDate:].

My work was to wrap -[NSRunLoop runUntilDate:] into something easy for unit tests.

```objc
@interface JLTWaiter : NSObject

@property (atomic, getter=isFinished) BOOL finished;
@property (atomic) NSTimeInterval waitQuantum;

- (void)waitWithTimeOut:(NSTimeInterval)timeOut;
- (void)waitWithTimeOut:(NSTimeInterval)timeOut
   atQuantumPeformBlock:(void(^)())block;

+ (JLTWaiter *)waiter;
+ (JLTWaiter *)waiterWithWaitQuantum:(NSTimeInterval)waitQuantum;
- (id)initWithWaitQuantum:(NSTimeInterval)waitQuantum;

@end
```

The flow is instantiate a waiter, have the waiter wait, tell the waiter when you finish.

```
- (void)waitWithTimeOut:(NSTimeInterval)timeOut
{
    [self waitWithTimeOut:timeOut atQuantumPeformBlock:NULL];
}

- (void)waitWithTimeOut:(NSTimeInterval)timeOut atQuantumPeformBlock:(void (^)())block
{
    NSDate *timeOutDate = [NSDate dateWithTimeIntervalSinceNow:timeOut];

    if (block) block();
    while (!self.finished && [timeOutDate timeIntervalSinceNow] > 0) {
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate dateWithTimeIntervalSinceNow:self.waitQuantum]];
        if (block) block();
    }
}

+ (instancetype)waiter
{
    return [[JLTWaiter alloc] init];
}

+ (instancetype)waiterWithWaitQuantum:(NSTimeInterval)waitQuantum
{
    return [[JLTWaiter alloc] initWithWaitQuantum:waitQuantum];
}

- (instancetype)initWithWaitQuantum:(NSTimeInterval)waitQuantum
{
    self = [super init];
    if (self) {
        _waitQuantum =  waitQuantum;
    }
    return self;
}

- (instancetype)init
{
    return [self initWithWaitQuantum:0.1];
}
```

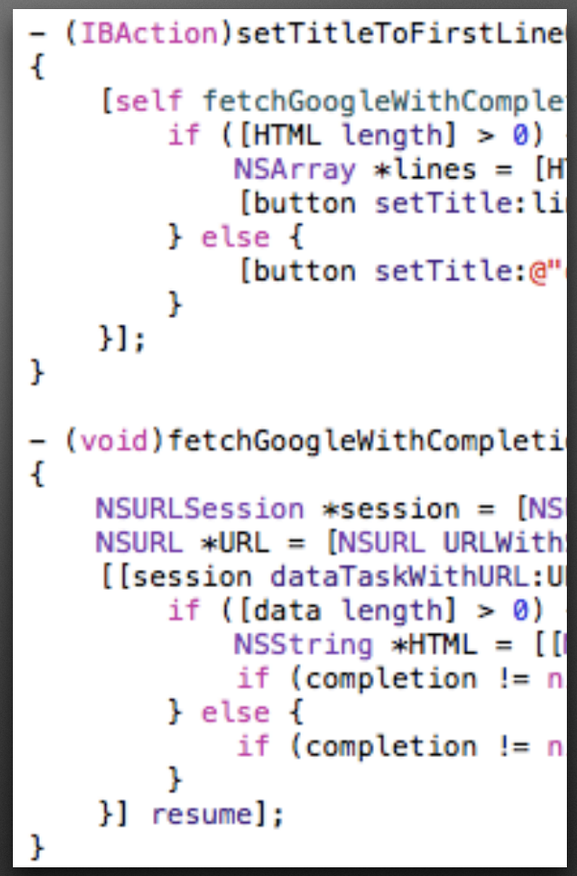
The core idea is a small class to wrap a more complex operation.

```objc
- (void)testWaiterSuccess
{
    JLTWaiter *waiter = [JLTWaiter waiter];
    [self.viewController swapHeightAndWidthWithDuration:2.0 completion:^{
        waiter.finished = YES;
    }];

    [waiter waitWithTimeOut:2.5];

    XCTAssertTrue(waiter.finished);
}

- (void)testWaiterFailure
{
    JLTWaiter *waiter = [JLTWaiter waiter];
    [self.viewController swapHeightAndWidthWithDuration:2.0 completion:^{
        waiter.finished = YES;
    }];

    [waiter waitWithTimeOut:1.5];

    XCTAssertFalse(waiter.finished);
}
```

Now, waiting for an animation and checking to see if the animation completed is easy.

Testing the Middle

The Problem of Remote Interoperability



Often, web services are developed in parallel with the client app.

As changes are made to both client and server, defects can occur which will not be caught by unit tests on the server or by mocked responses on the client.

Special web service testing must be done ensure interoperability.

Web Service Testing

- Specialized Functional Tests
  - Advanced REST Client
  - SoapUI

The usual solution for web service testing is specialized functional tests.

What's Wrong With That

- Separate Tool Chain
- Separate Automation System
- Duplication of Effort to Implement API
- Functional Test Must Match Client

The real killer here is keeping the functional test in line with the app's use of the web service.

Using Unit Testing Tools to Test a Web Service

- Client Unit Tests Without Mocking the Response
  - It's the Same Tool Chain
  - It's the Same Automation System
  - No Duplication of Effort
  - Unit Test Tool Uses the Client; Never a Mismatch

So why not use the client's unit testing framework.

What's Wrong With That

- It's Not Really a Unit Test
- Network Transactions Require an Async Task
- Network Requests Might Not Get a Response

The first one is really a name thing… who cares what it's called.

I ended up calling them partial system tests.

The other two bullet points look really familiar.

JLTWaiter Goals

- Pause Execution of a Unit Test to Wait for an Async Task
- Continue Execution When the Async Task Completes or After a Specified Timeout
- Determine If the Async Task Completed

Hey, that's right, I've already solved this problem.

What did I say before… "Simple Generic Solutions are Flexible".

```
- (void)testGoogleHomepage
{
    JLTViewController *target = self.viewController;
    JLTWaiter *waiter = [JLTWaiter waiter];

    [target fetchGoogleWithCompletion:^(NSString *HTML, NSError *error) {
        waiter.finished = YES;
        XCTAssertNotNil(HTML);
        XCTAssertNil(error);
    }];
    [waiter waitWithTimeOut:5.0];

    XCTAssertTrue(waiter.finished);
}
```

It feels like an ad-hoc solution, but I've used it for a couple of years now and it's proven itself to me as a straightforward reliable solution.

Recap

- Tiny Solutions Presented
  - Polyfill to Make a Needed API (JLTMapFilterReduce)
  - Provide Extra Data (JLTContextInfo)
  - Reduce Boilerplate Code (JLTEasyObserving)
  - Wrap a More Complex Operation (JLTWaiter)
- Thinking Small Can Change Your Code for the Better

With some extra thought and some extra work, you can fix your pain points by building custom reusable bits of code. They can be as abstract or as specific as you need them to be.

In talking with people beforehand, I had a hard time describing this talk. For a talk about distilling the core functionality, it's odd that I couldn't distill it's core functionality. To me, the lesson is this doesn't work for everything.

I hope you've enjoyed the presentation.

Does anyone have questions?