MVVM pattern in iOS development

Denis Lebedev,
iOS Hipster @ Wargaming

mailto: d_lebedev@wargaming.net
linkedin: http://linkedin.com/in/dlebedev

WARGAMING.NET

LET'S BATTLE

Agenda

- Wargaming Service Mobile Apps dept.
- MVC
- MVVM
- Development hints
- Reference

WARGAMING SERVICE MOBILE APPS

SMA department

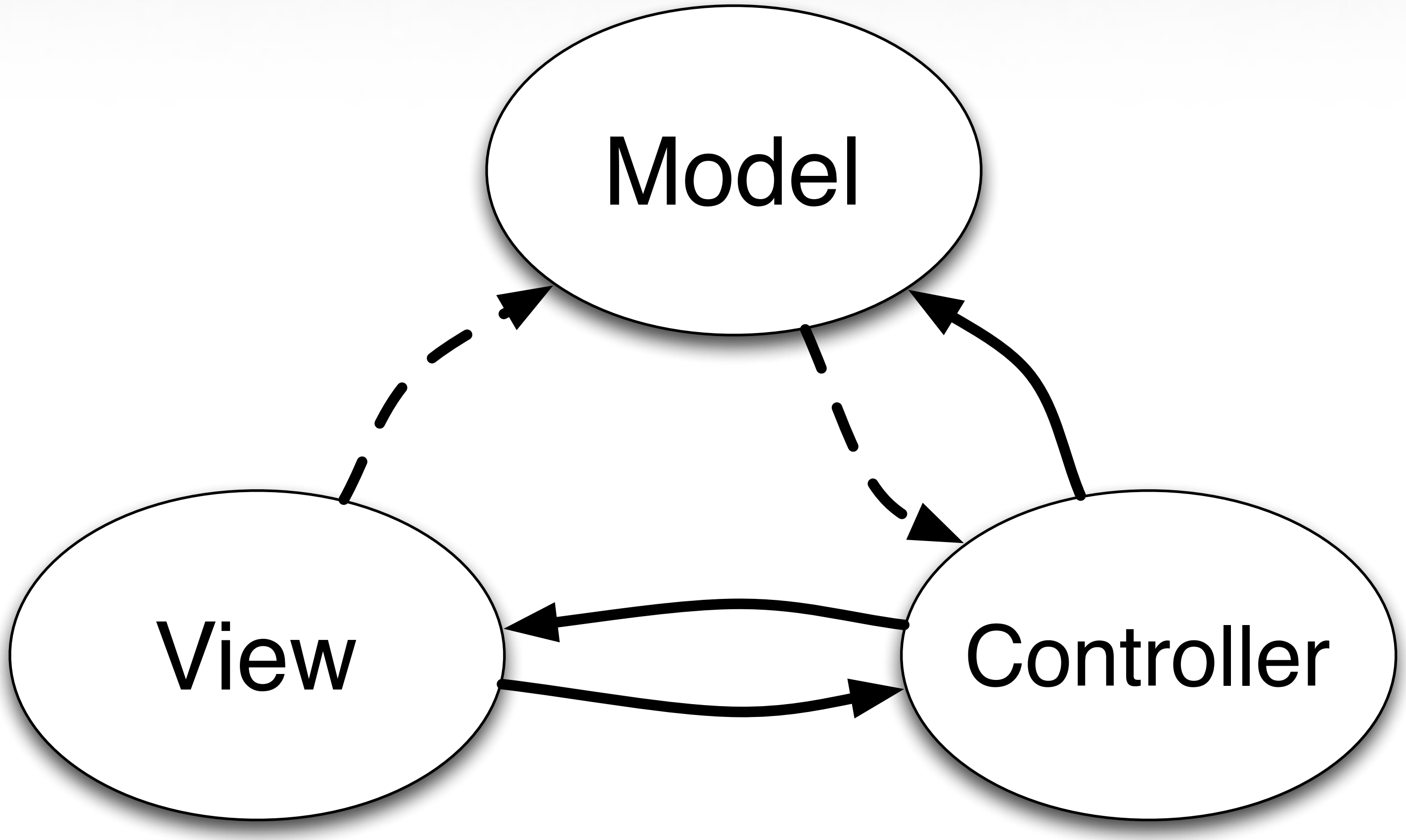
- We make apps for user engagement
- Quality really matters
- We experiment a lot with new tools & technologies

Apps

WARGAMING.NET
LET'S BATTLE

MODEL-VIEW-CONTROLLER

MVC Diagram



Classical principles

- Loose coupling
- Model is decoupled; view/controller can listen to it’s changes
- View knows about model, can manipulate controller
- Controller knows everything and “does all the things” ©

View Controller responsibilities

- Owns & manages view’s state
- Acts as the datasource for tableviews
- Maps model’s properties to view values
- …

<u>MASSIVE</u> VIEW CONTROLLER

Real world MVC

- Controllers are ‘almost’ not reusable
- Controllers are large
- View is very tight to it’s controller
- Controllers are hard to unit test

Solving Massive Controller problems

- Separate object acting as tableview dataSource
- ConverterHelper, ‘MyFancyAttributeToStringHelper’, etc..

Testing

- Unit tests of controllers code is pain
- Sort of UI tests (Calabash, KIF) is required

Wrapping up

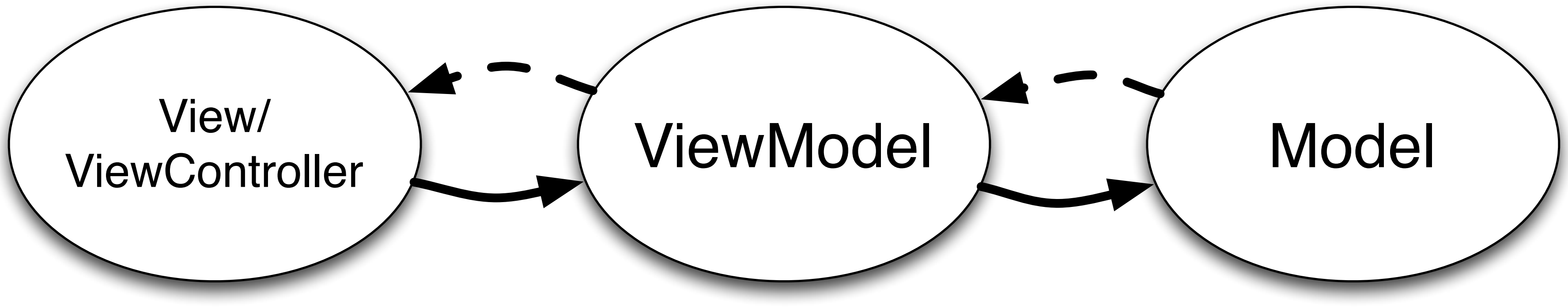
MVC is not bad at all.

Wrapping up

What can we do for better world?

MODEL-VIEW-VIEWMODEL

MVVM Diagram



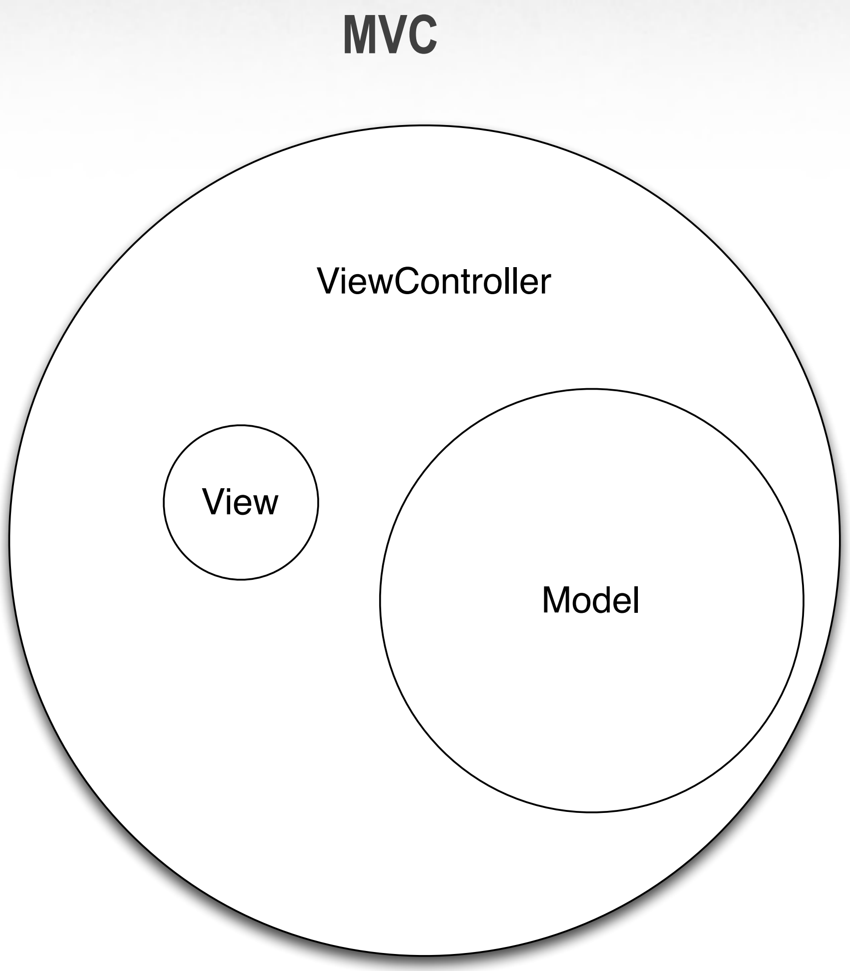
MVVM Diagram

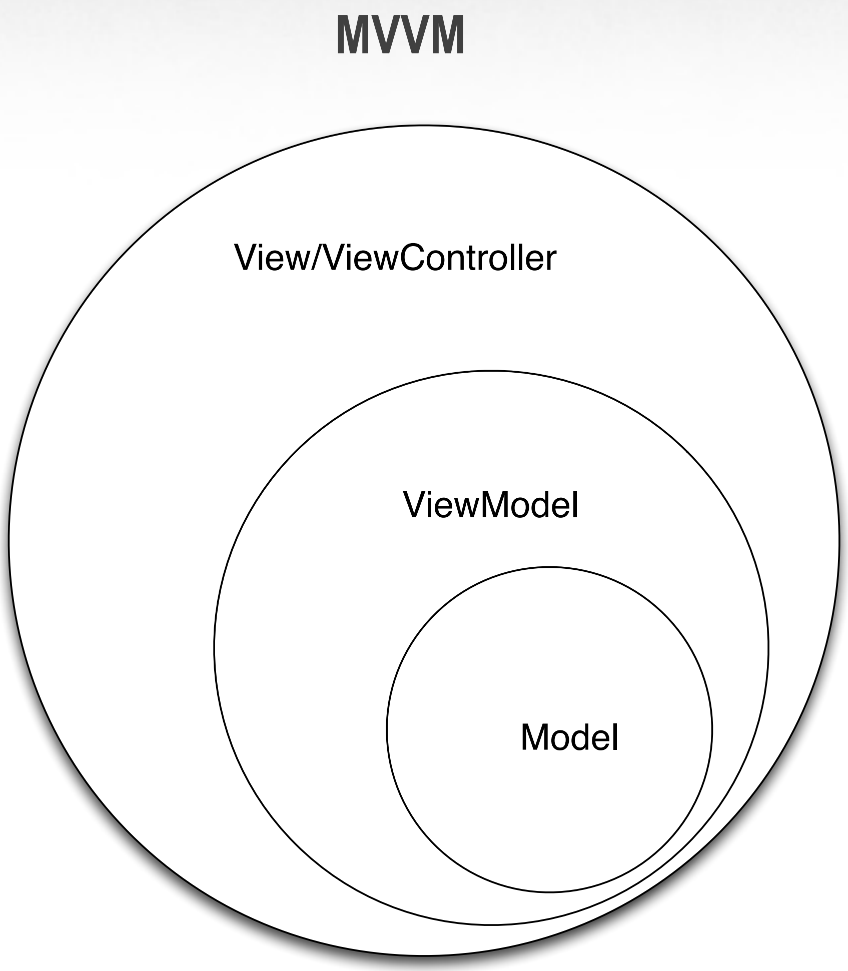
Wait what? It’s the same!

Key differences with MVC

- ViewController owns it’s view model
- View model doesn’t know anything about controller

Key differences with MVC





MVVM Models

- Can represent single domain entity or collection
- Responsible for fetching/storing data

MVVM Views

- Cocoa note: MVVM view = view + view controller
- view is stateless
- views/view controllers become smaller

MVVM View Models

- encapsulates all view data/properties and binds them to view
- validation logic
- actions

ReactiveCocoa

- Functional reactive paradigm in Cocoa world
- Fancy bindings
- Composes of sync/async events of any kind
- Mature, production-ready framework

ReactiveCocoa

```
RAC(self, title) = RACObserve(self, viewModel.nickname);
```

ReactiveCocoa

```
RAC(self, stringType) = [RACObserve(self, model.type)

   map:^id(NSNumber *v) {
      return v.intValue == 0 ? @"zero" : @"non-zero";
   }];
```

Benefits

- We don’t need to to test the UI (actually we should)
- We <u>can</u> implement app logic without any UI
- view model is easy testable
- view model are (almost) platform independent

Platform agnostic code

- iPad / iPhone / OS X code sharing
- MVVM + Xamarin = shared Windows / OS X code

How we came to MVVM

- Already familiar with ReactiveCocoa
- Strong need in good internal testing
- Fresh project developed from scratch

DEVELOPMENT HINTS

What actually does view controller?

- Layout
- Animations
- Device rotation
- View transitions

- All sensible state is stored in view model

Instantiation

- Every controller has *viewModel* property
- Some views may have separate view models
- Inject view model to controller during instantiation
- Inject view model's important properties during instantiation

ViewModel interface

- *RACSignals* as model properties where possible
- *RACSignal* for data requests
- Model property (NSArray, domain object, etc.)

WPAFeaturesViewModel.h

```
@interface WPAPlaneFeaturesViewModel : NSObject

@property (copy, nonatomic) NSArray *planeRows;

@property (strong, nonatomic, readonly) RACCommand *forwardCommand;
@property (strong, nonatomic, readonly) RACSignal *forwardHidden;

@property (strong, nonatomic) RACSignal *nextPlaneTitle;

- (instancetype)initWithReferencePlane:(WOWPPlane *)plane
                  classMates:(NSArray *)planes;

@end
```

ViewModel for tableviews

- ViewModel has “rows” property of type NSArray
- Row is some NSObject subclass with data very coupled to cell
- Formatters, etc. are applied to row class, not cell
- Controller binds to “rows” and reloads data after property changes

›WPAPlaneRow.h

```
@interface WPAPlaneRow : NSObject

@property (strong, nonatomic) WPAFeature *feature;
@property (copy, nonatomic) NSString *value;
@property (copy, nonatomic) NSString *grade;
@property (copy, nonatomic) NSString *referenceGrade;

@end
```

Testing

ViewModel tests are very obvious and atomic:

- State of model can be determined in any moment
- Property change affects view model internal state
- *RACCommand* changes view model state
- *RACSignal* pushes new state to subscribers

Real world testing

- Unit tests for view models
- Integration tests (for controllers) with KIF

IMPERATIVE VS DECLARATIVE

How to be functional and reactive

- Write declarative code instead of imperative
- Describe how properties are related to each other

Functional Reactive view model

- Almost all code is in constructor
- Describes how the view *should* work in terms of commands and properties

Functional Reactive view model

- *The login button can be pressed when username and password aren’t blank*
- *The error should be cleared after 5 seconds it was displayed*

REFERENCE

Functional Reactive view model

- ReactiveCocoa https://github.com/ReactiveCocoa/ReactiveCocoa
- ReactiveViewModel https://github.com/ReactiveCocoa/ReactiveViewModel

- FRP on iOS https://leanpub.com/iosfrp

- FunctionalReactivePixels https://github.com/AshFurrow/FunctionalReactivePixels
- MVVMExample https://github.com/garnett/MVVMExample

THANK YOU!