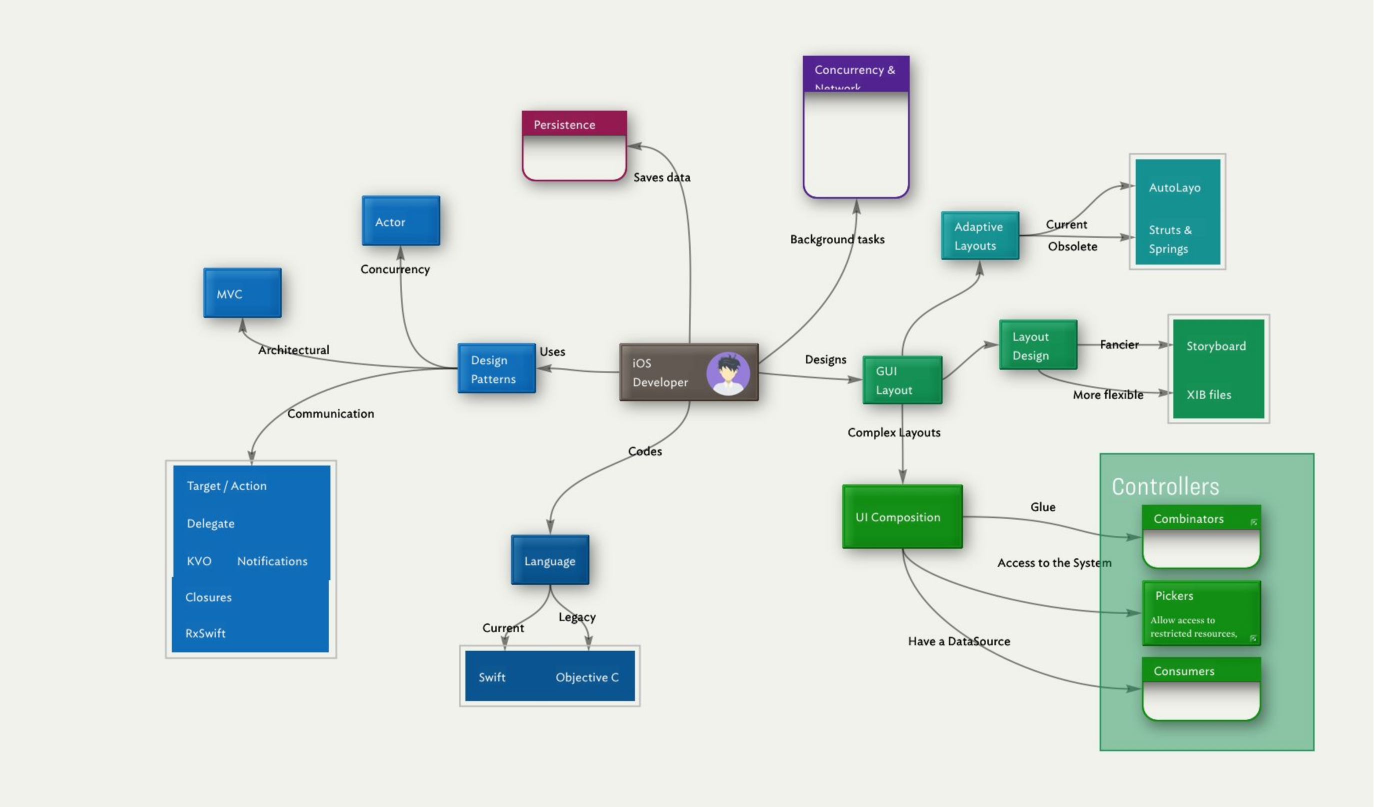
iOS Foundations With Swift 4.2

Overview Of iOS Development

- Environments & Devices
    - iOS
    - macOS
    - tvOS
    - watchOS
    - Linux

Knowledge Map of an iOS Developer



iOS Developer

Uses

Design Patterns

Architectural

MVC

Concurrency

Actor

Communication

Target / Action

Delegate

KVO

Notifications

Closures

RxSwift

Saves data

Persistence

Background tasks

Concurrency & Network

Codes

Language

Current

Legacy

Swift

Objective C

Designs

GUI Layout

Adaptive Layouts

Current

Obsolete

AutoLayo

Struts & Springs

Layout Design

Fancier

More flexible

Storyboard

XIB files

Complex Layouts

UI Composition

Glue

Access to the System

Have a DataSource

Controllers

Combinators

Pickers

Allow access to restricted resources,

Consumers

What you’re going to learn and why

- Swift 4.2. The new *de facto* language.
- MVC (Model View Controller): The cornerstone of every iOS App.
- Archiving: A simple way of decoding and encoding objects in JSON and persisting them.
- UI Composition: Create UIs by glueing simple components.
- Sending Information among objects: target-action, delegate, closures, notifications.
- Xib Files: The most flexible way to create reusable components.
- AutoLayout: standard way of creating adaptive layouts.
- **The purpose of this course is to build Solid Foundations.**

An Overview of iOS

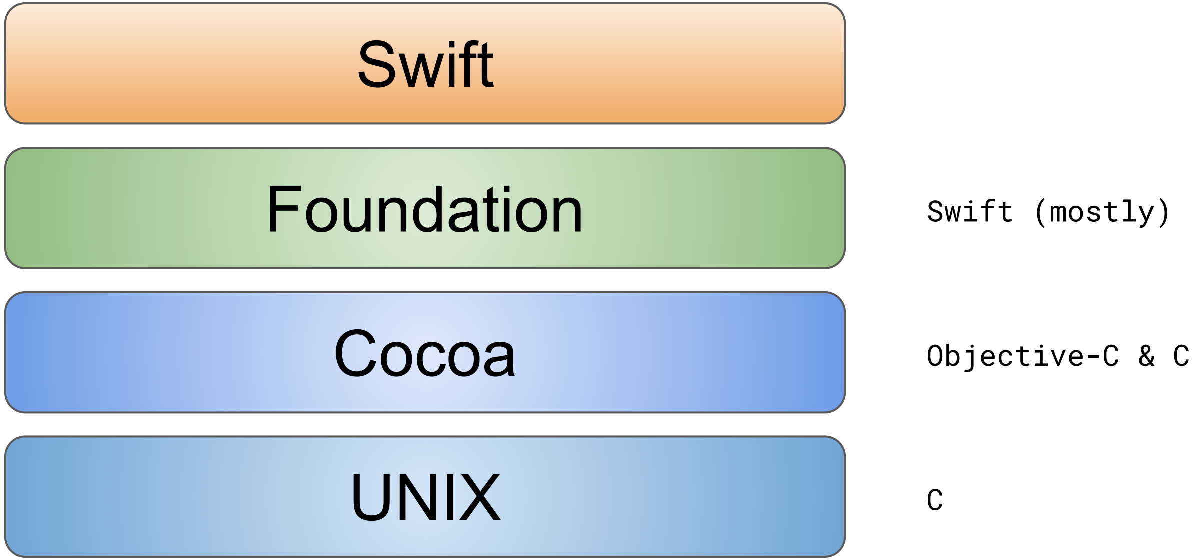
- It’s UNIX!
- MicroKernel BSD, very similar to macOS
- File permissions are very restrictive. All apps are enclosed in a folder called the *Sandbox*.
- The OS constantly checks the usage of certain resources, such as memory and file space. Apps that misuse them will face drastic measures.
- The file system is *APFS*: optimized for flash drives.

iOS doesn’t run on a server!

TANSTAAFL: There ain't no such thing as a free lunch.
Heinlein. “The moon is a harsh mistress”

- Never forget the limited capabilities of iOS devices.
- Limited resources:
  - Memory
  - Battery
- Have as few objects in memory as possible at any given moment.
- Battery killers: GPS, antenna, background operations.

The Programming Stack



Swift

Foundation — Swift (mostly)

Cocoa — Objective-C & C

UNIX — C

Tools

- **Language:** Swift 4.2 Statically typed, functional / OO mix. Easy to learn, hard to master.
- **IDE:** Xcode, maybe AppCode.
- **Frameworks:** Foundation, UIKit, GCD, CoreAnimation, CoreData, CoreML, CoreLocation, MapKit, CoreImage, CoreAudio, etc…
- **TDD:** Test Driven Development
- **DDD:** Domain Driven Development
- **Dash:** Documentation browser
- **Tinderbox & XMind:** Notes & brainstorming.
- **Homebrew:** apt-get for macOS
- **CocoaPods:** Dependency management

A Tour of Swift

Playground

Getting your feet wet

A Storm of Sounds. Your first App!

What have we done wrong?

- Everything.
- What is this view controller?
- What’s a Storyboard? Who creates it?
- What’s the lifecycle of the App? How does it start?
- What in the name of The Seven are those IBOutlets and IBActions?
- We started by the GUI. That’s the part of the App that is most likely to change. This is akin to building a house starting by the roof.
- The App lacks an architecture that would allow it to grow.
- There’s a lot of magic going on, and we have little or no understanding of what’s going on.

you know nothing,
jon snow.

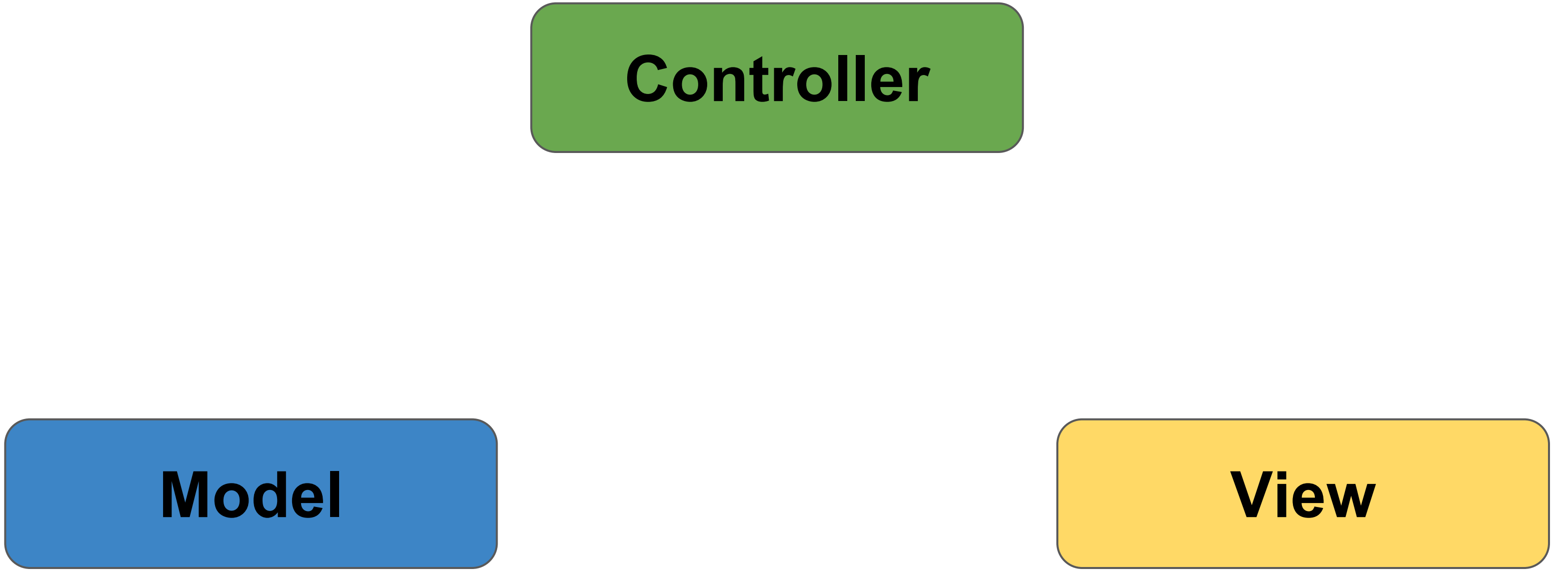
The Foundation of Every App

The Model-View-Controller Design Pattern

What is a Design Pattern?

- A battle-proven, optimal solution for a common problem.
- The MVC is an Architectural Pattern: it describes a sensible way of building our App as a whole.
- It was discovered at the Xerox PARC lab, in Palo Alto, decades ago.

MVC: Model View Controller



Controller

Model

View

The Model

- The essence of your software, what it truly is.
- Irrespective of the way it interacts with the user (web, phone, desktop, whatever).
- Examples of models:
  - A stock trading application
  - A word processor
  - A game

The Controller

The View

There’s more to the controller than meets the eye

Information Flow

Information Flow

An Example

MVC in Action: The Calculator App

Information Sharing Recap

- Target / Action
- Delegate
- Notifications
- Trailing closures

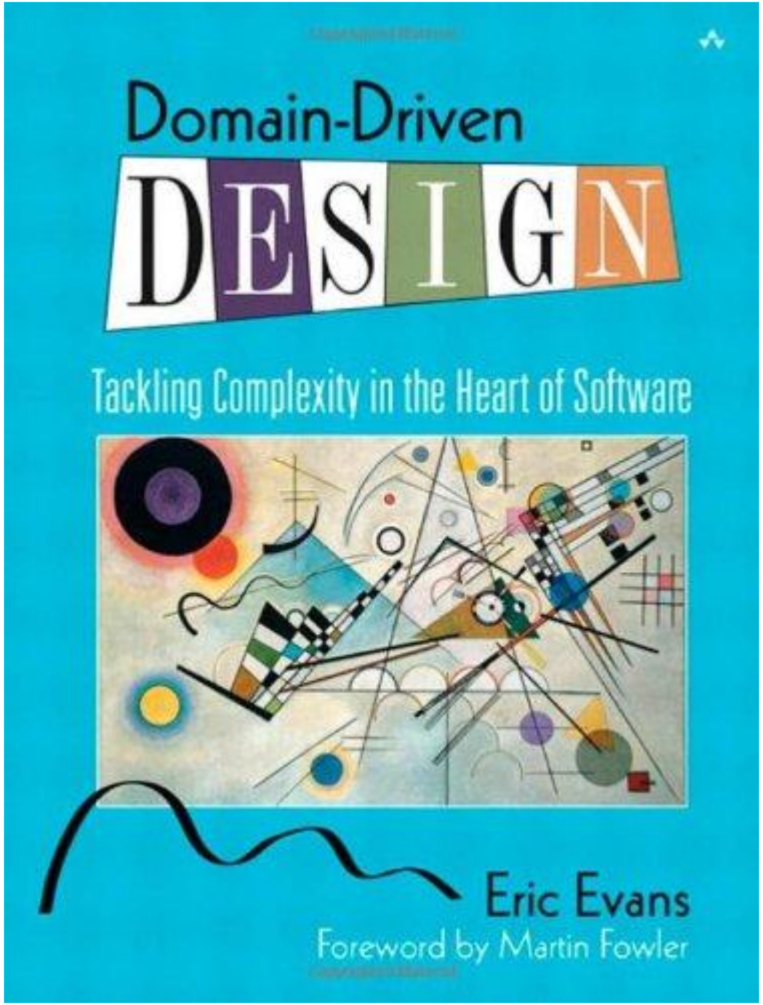
Our First App: Westeros

- How to design a complex App from scratch while making sure Hell is not a demo of your work.
- We will use 2 techniques
    - DDD by Eric Evans
    - TDD by Kent Beck

Domain Driven Design

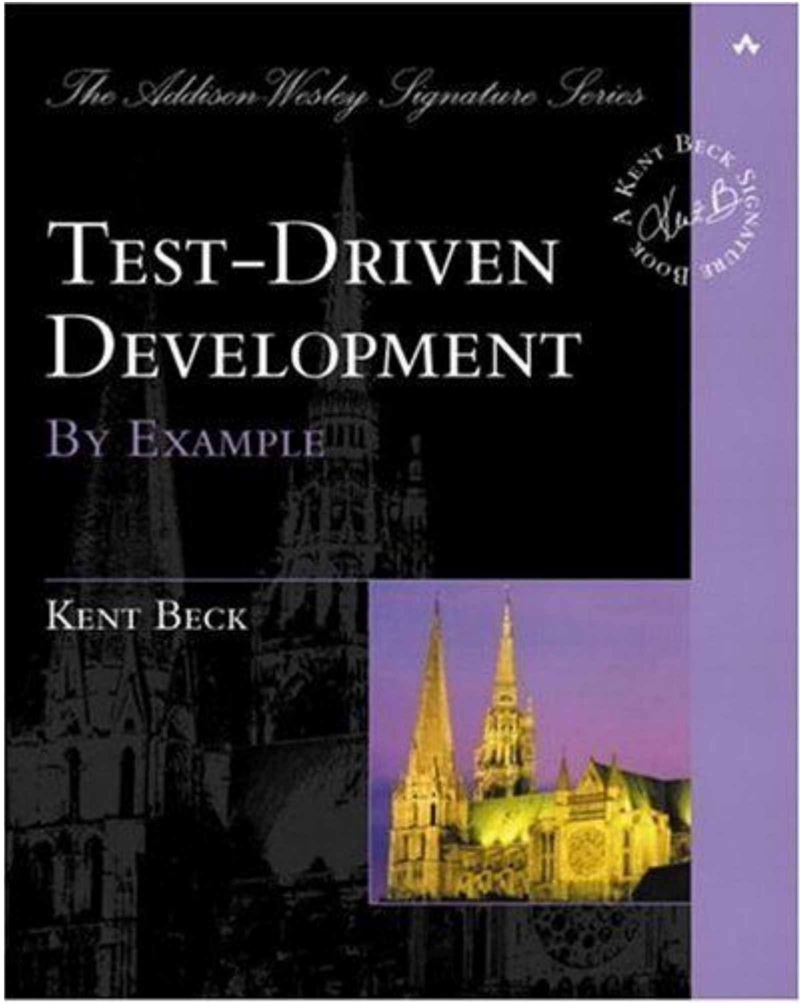
- The primary focus is the *domain*.
- The subject area to which the user applies a program is the *domain* of the software.

Start with the Model and build a solid core before moving to the Controllers and Views.


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Test Driven Development

- Design the specification of a *small* feature *before* implementing it.
- Only then *write the code necessary to pass* the test of the specification.
- *Never* write code except for passing a test.

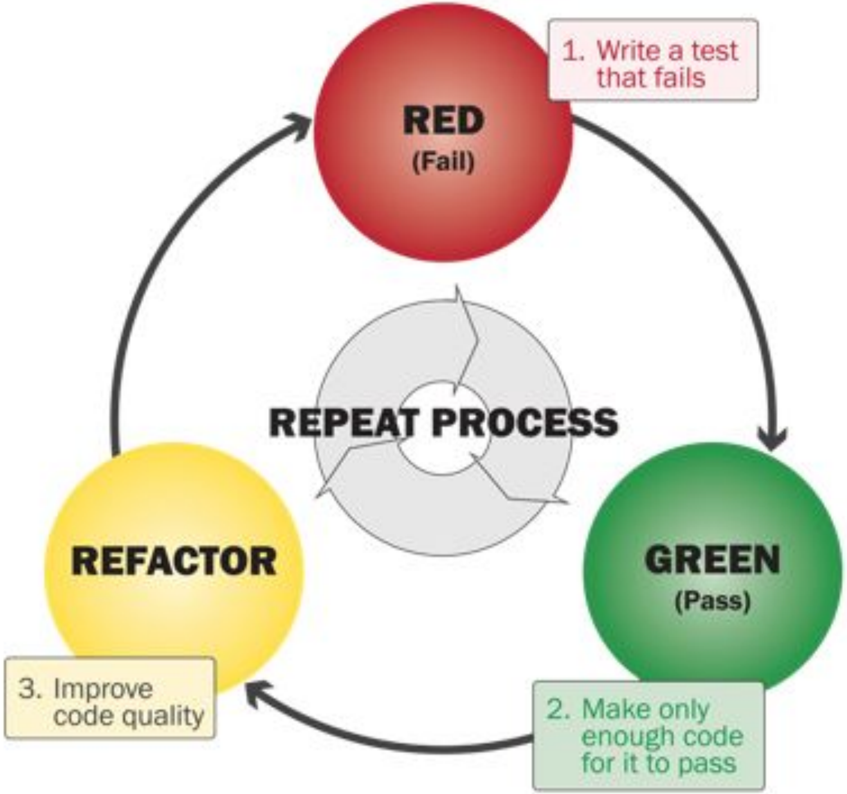

The Addison Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

Advantages of TDD

- Code is mostly guaranteed to work as defined
- Tackles complexity by working in small iterations
- Allows for optimal solutions to emerge
- Avoids situations where you end up “spinning your wheels”.
- Prevents over engineering and over thinking.
- Keeps you focused on what really matters.
- Goes well with some added “pomodoro” ;-)

The TDD Cycle

- Red
- Green
- Refactor


RED
(Fail)
1. Write a test that fails
REPEAT PROCESS
GREEN
(Pass)
2. Make only enough code for it to pass
REFACTOR
3. Improve code quality

Red: Write a letter to Santa Claus

- Write down the features that you *wished you had*.
- Never mind if you have no idea how to implement it...yet. This is dreamland, so *dream*.
- It doesn’t even compile? Good! You’re dreaming big!

Green: Write the **simplest** solution that works

- Is it a horrendous kludge?
- Good! We’ll fix it later.
- Seriously. :-)

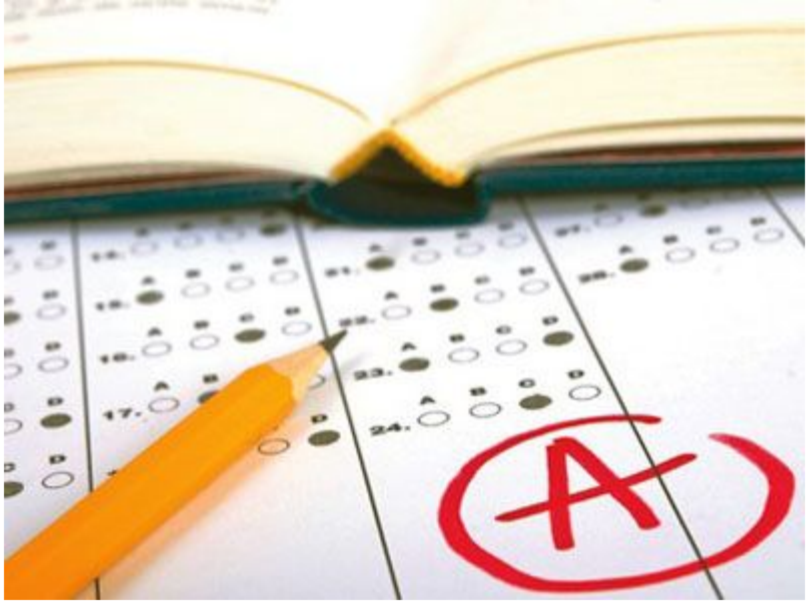
Refactor: remove all kludges

- Improve the quality of the code
- Make it general instead of specific.

Done? Now repeat the cycle!

What makes a good test?

- Fast.
- Tests only one thing.
- Repeatable & predictable.
- Only fails if **your** code is broken.
- **You** should only test the code **you've** written.



Westeros

- New project and get rid of all the magic: a *clean slate*.
- Your first MVC, free of charge

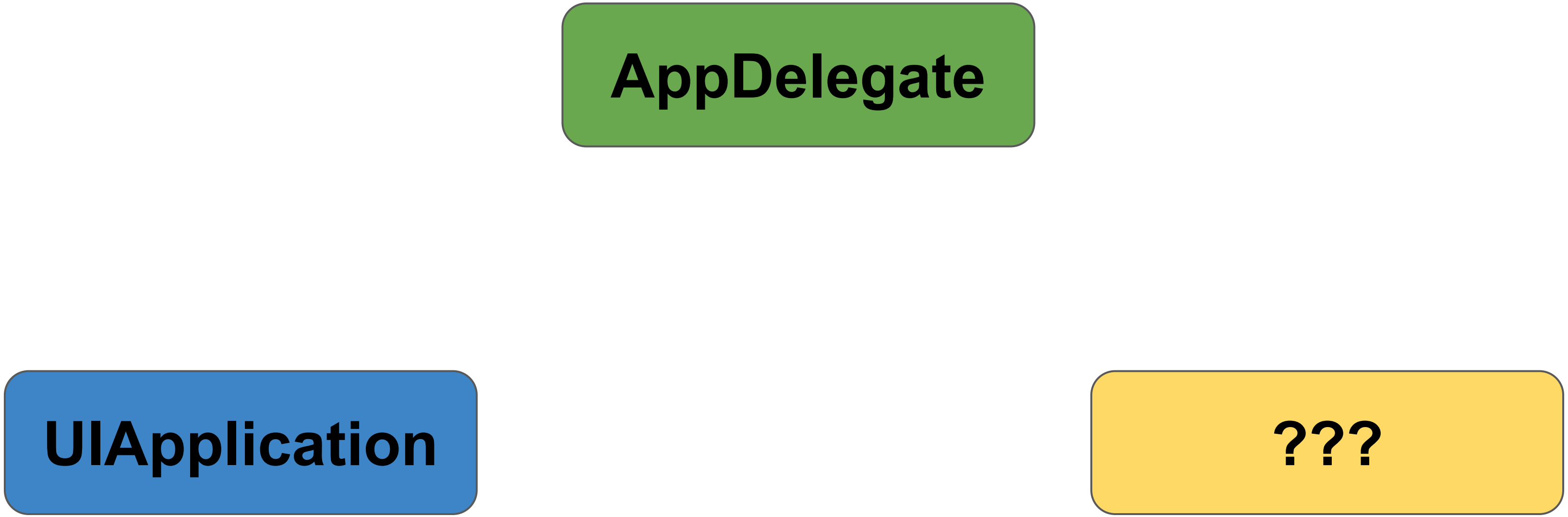
Empty App

What happens when we run the empty app?

- The system creates an object called UIApplication.
    - It represents our App.
    - The OS communicates with it.
- The system also creates an object called AppDelegate
    - It's a helper (delegate) for UIApplication
    - It allows us to respond to the information the OS sends to UIApplication
    - This is where we start adding behavior to our App.

Empty App

Does it ring a bell?



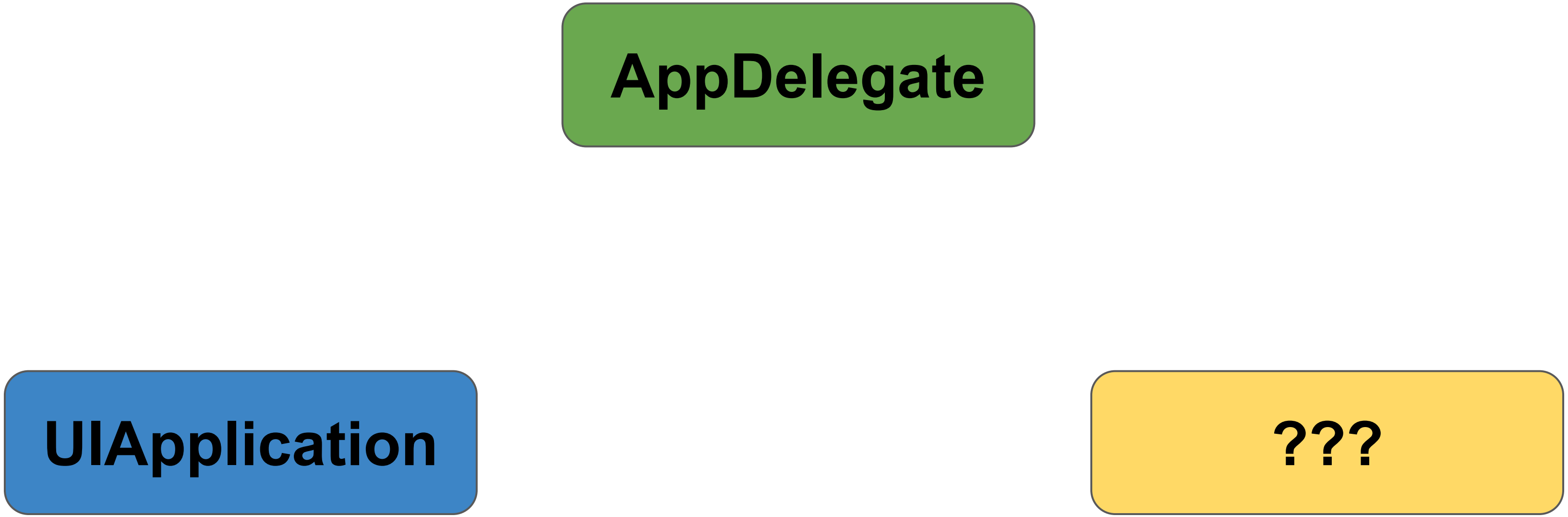
AppDelegate

UIApplication

???

Empty App: why is the screen black?

Because there’s no view! Let’s create one.

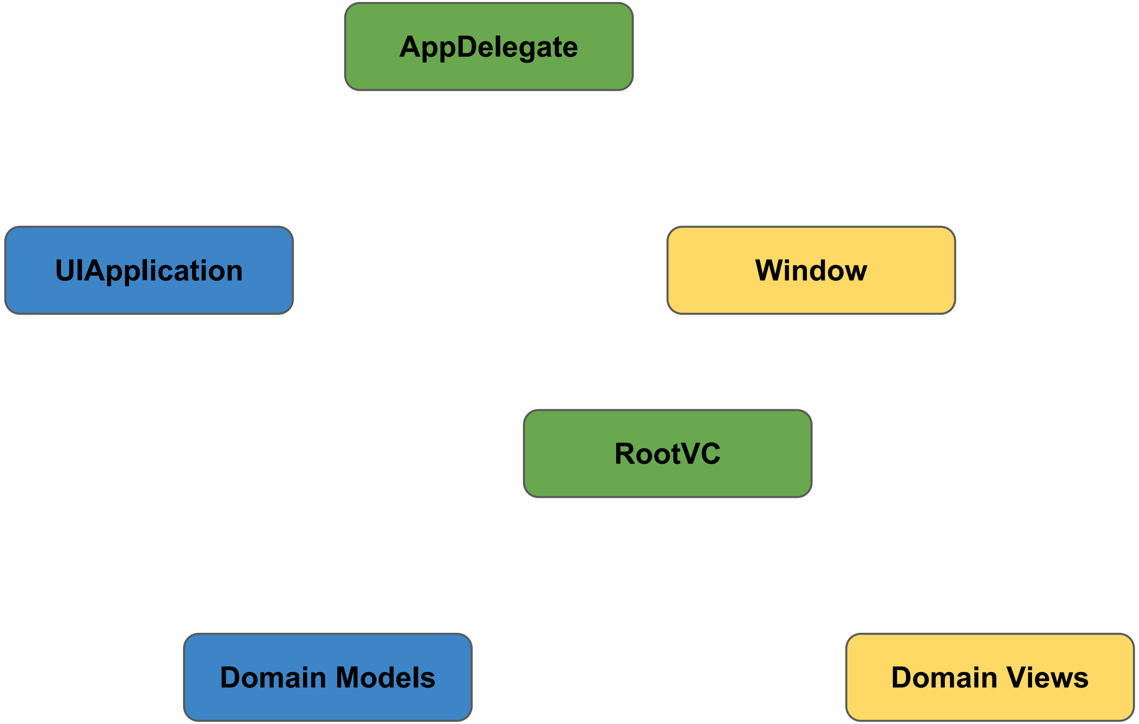


AppDelegate

UIApplication

???

Hierarchy of an App



AppDelegate

UIApplication

Window

RootVC

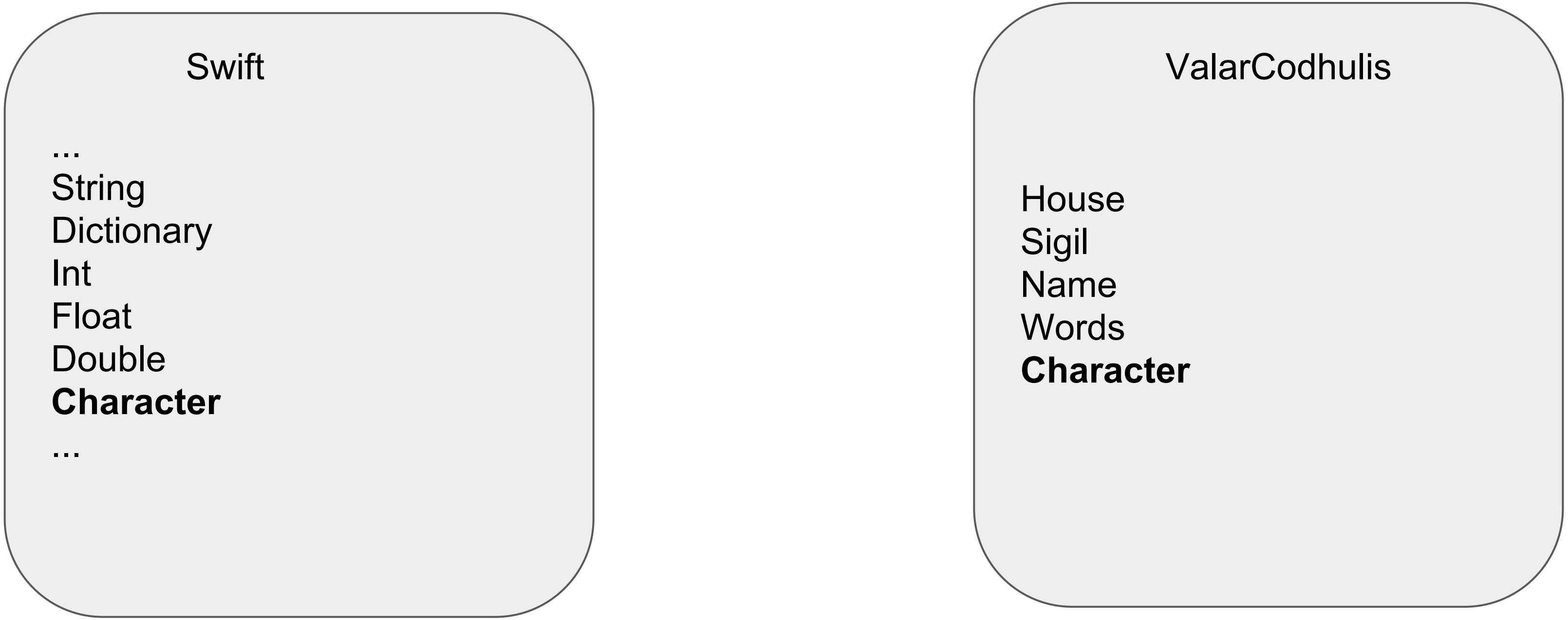
Domain Models

Domain Views

Let’s start with the domain

- What are we going to simulate?
- What are the main concepts?
- Write them down, so we find our domain.

A Clash of Characters



Swift

...
String
Dictionary
Int
Float
Double
Character
...

ValarCodhulis

House
Sigil
Name
Words
Character

A Clash of Characters

To make sure the compiler knows which Character we mean, we should prepend the name of the module...always.

Swift.Character

Westeros.Character

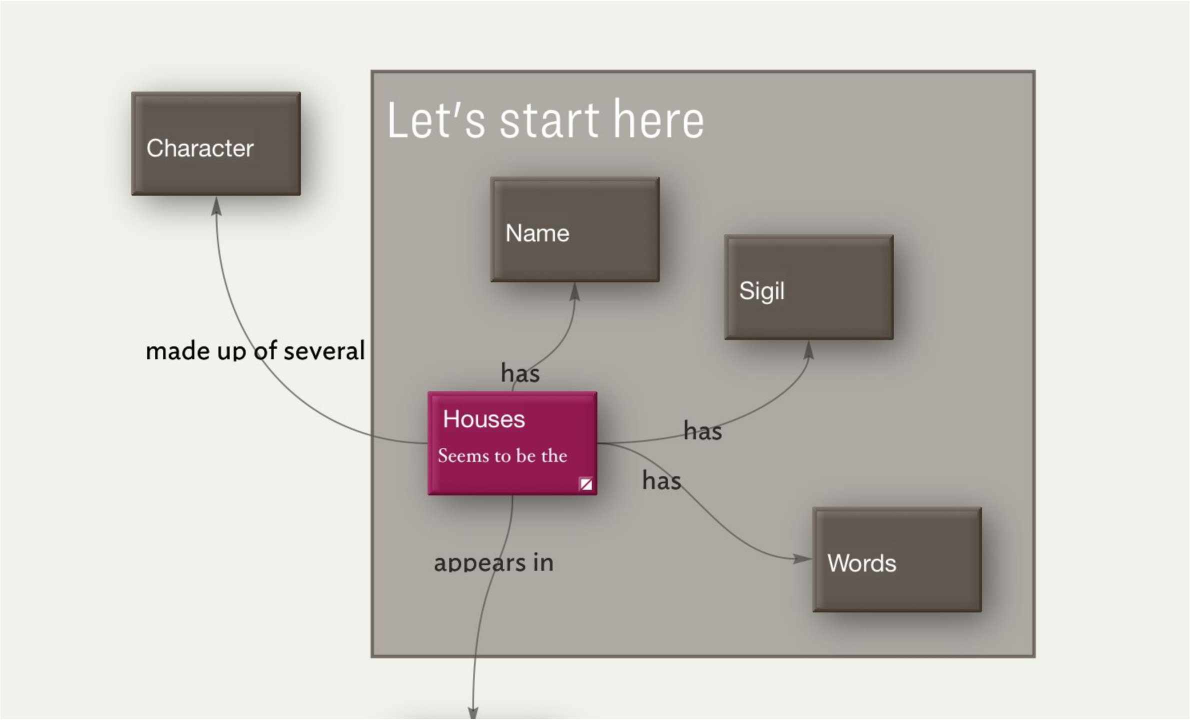
Should we implement Comparable and Equatable?

- NO!
- We don’t need it now. So add it to “Some day maybe” and forget about it.
- Leave tomorrow’s stuff for tomorrow.

A House has several members

- We forgot about that!
- Let’s add a test for adding Persons and counting the persons in a House.

The Core of our Domain



HouseViewController

UIViewController

Anti Patterns

- Design Patterns are known solutions to common problems.
- Anti-patterns are known blunders to common problems.
- Let’s check 2 of the most common in iOS
  - God Class
  - Class Explosion

© Limebridge 2011
WELL THE MARKETING FOCUS GROUP GAVE POSITIVE FEEDBACK IN ALL THE AREAS THEY WERE ASKED ABOUT, SO I'M NOT SURE WHY IT'S NOT SELLING.
KUDELKA.

God Class

One Class to rule them all, and in the darkness bind them.

- A "God Class" is an object that controls way too many other objects in the system.
- Has grown beyond all logic to become The Class That Does Everything.
- Described in “Object Oriented Heuristics” by Arthur Riel

God Class in iOS

- Very common error.
- Usually a UIViewController.
- Many iOS developers mistakenly take it for a bug in the MVC, calling it the *Massive-View-Controller*.
- It has **nothing** to do with the MVC.

Class Explosion

- The opposite of a God Class: there are way too many classes to perform a task.
- The system is hard to understand, and difficult to extend.
- Usually caused by the dogmatic application of a poorly though architectural design patterns, such as VIPER.

💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩
💩💩💩💩💩💩💩💩💩💩

Class Explosion

God Class

VIPER

Massive View Controller

Virtue Lies in the Middle Ground

MVC

VIPER

MVVM

Massive View Controller

Don’t dismiss simplicity. Simple means solid.

The entire mansion of mathematics was erected on a foundation of this kind of irreducibly simple, yet logically rock-solid, axiom.

Cixin Liu - The Dark Forest.

What is really a Controller?

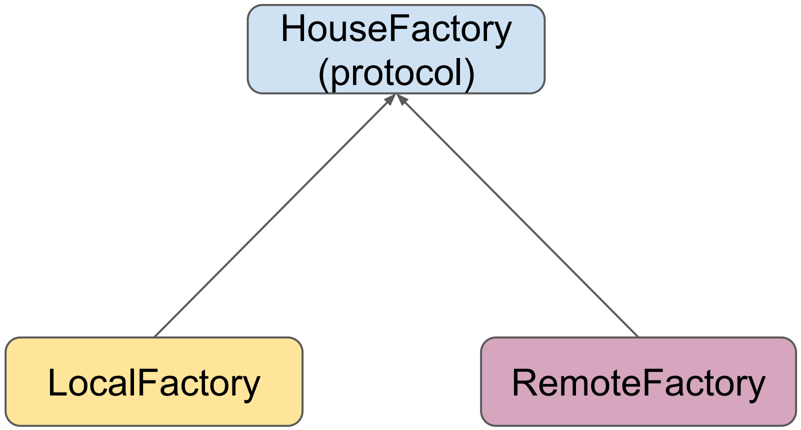
Combinators

How far have we come

Check the knowledge map

Factory Design Pattern

- Hides the details of object creation
- Centralizes in a single place the creation of objects.
- Allows for different object creation strategies.
- The rest of your App doesn't need to know where the objects are coming from.



Creates House instances by reading from a local file.

Creates House instances from a remote web service, via a REST API.

Singleton Design Pattern

- Ensure that only one instance of a class is created.
- Factories are usually a Singleton.
- In Swift, Singletons are implemented as a static property.

Singleton Design Pattern in Swift

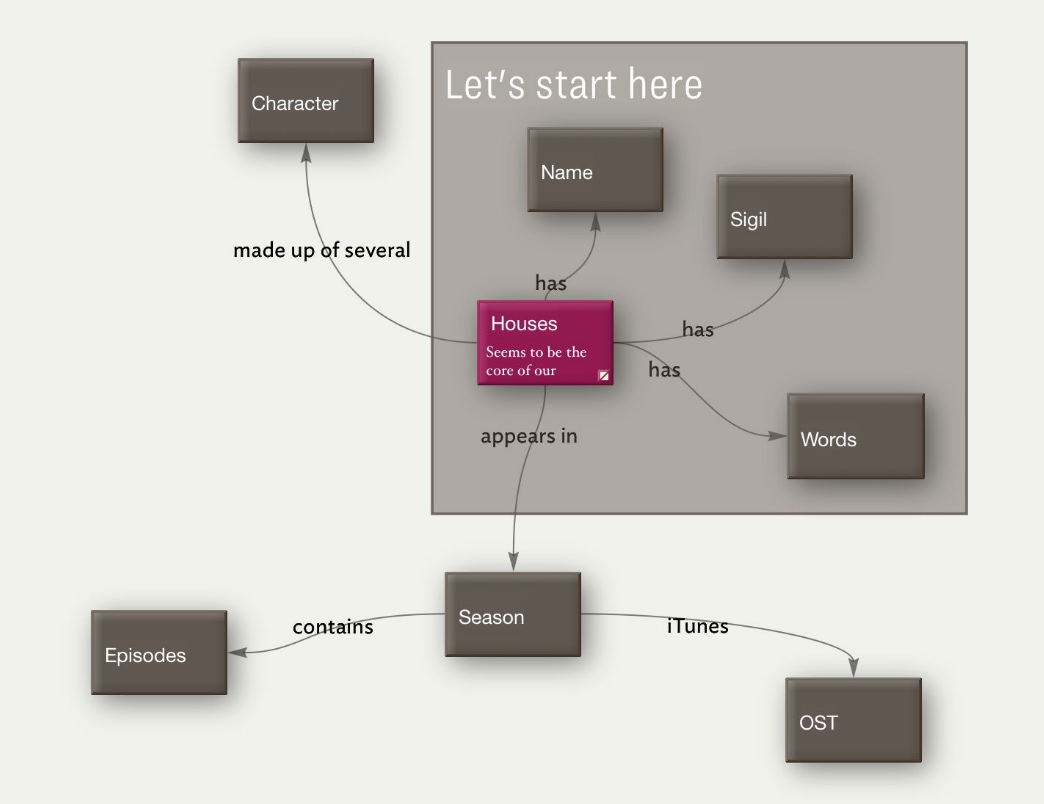
```swift
final class Repository{
    static let local : HouseFactory = LocalFactory()

    private init(){} // make sure no one can create an instance
}
```

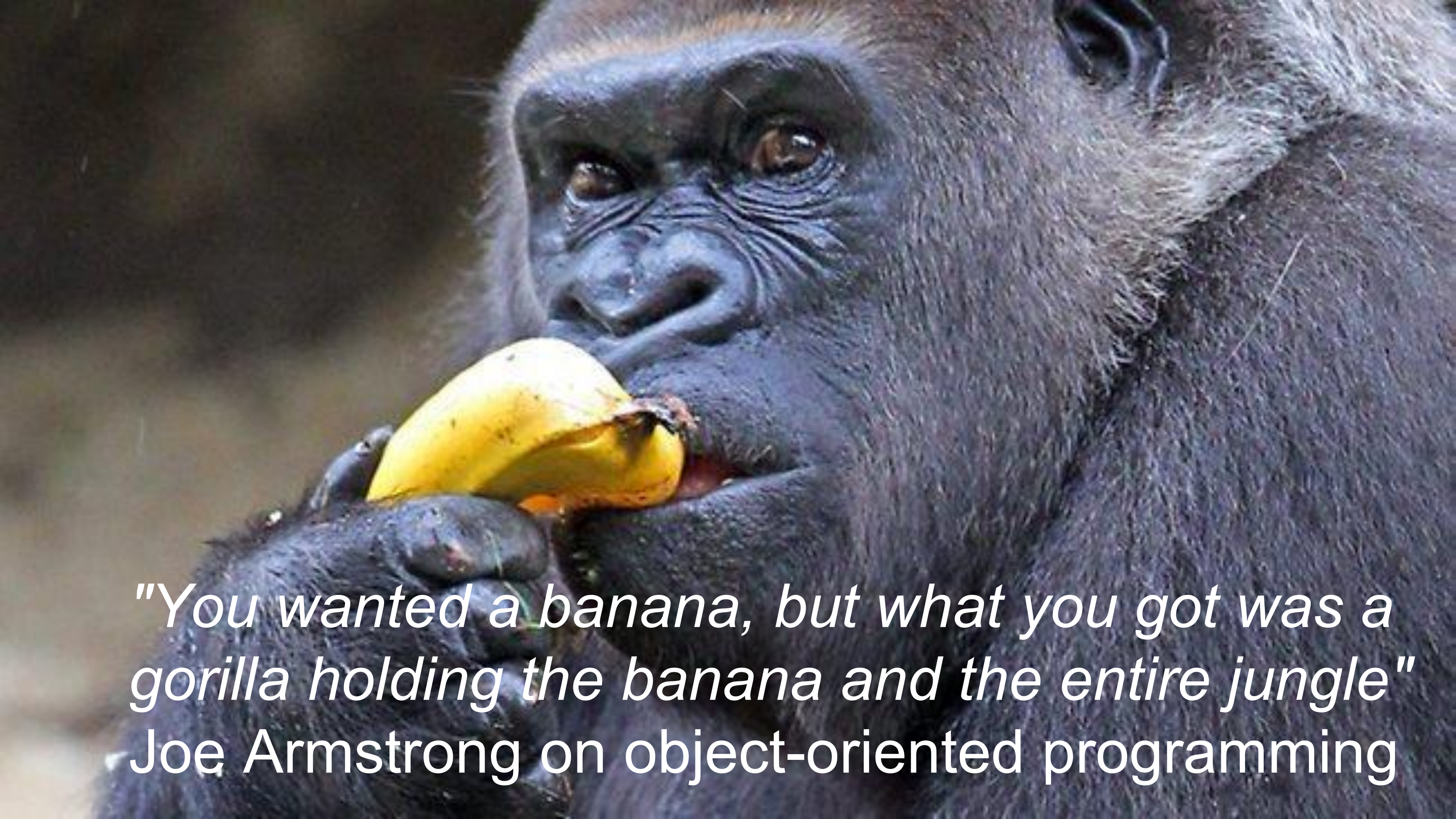
<CODE>

Limitations of the Model

A large graph or objects...all in memory.

- Each House requires several Persons, Names, Sigils and Words
- On a device with limited memory, this is BAD.





"You wanted a banana, but what you got was a gorilla holding the banana and the entire jungle"
Joe Armstrong on object-oriented programming

Limitations of the Model

No guarantee of single representation.

- There could be several objects in memory representing the same House.
  - More memory issues
  - Consistency issues.

Repetitive mindless tasks...

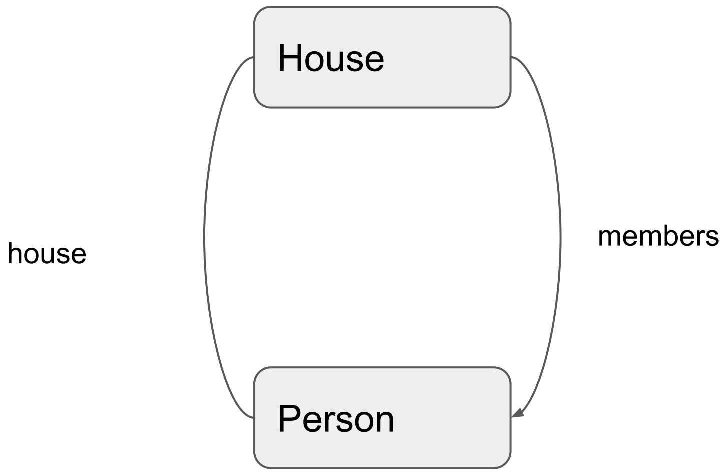
When we create a person, we must provide it with it's House.

Then, we must add it to the House.

This is idiotic.

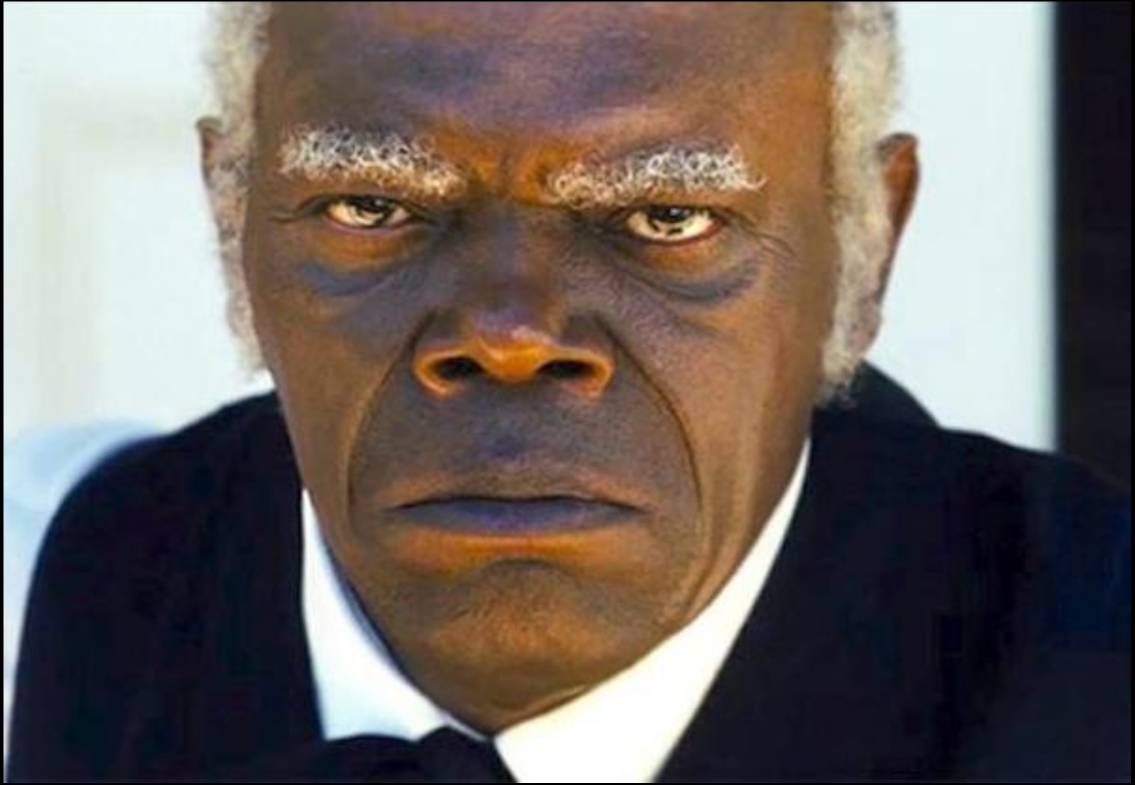
Whenever setting any of the 2 properties (members or house), the other one should be automagically updated.

We can do by hand, but it sounds like...



LeQuint Dickey’s Mine for Developers!

(...) all day, every day, you will be swingin' a sledgehammer, turnin' big rocks into little rocks.



Never. Lose. Hope.

There are tools that allow us to

- Keep memory usage as low as possible.
- Manage complex graphs of objects
- Keep relationships between objects always in a correct state...without grunt work.
- **Core Data**
- **Realm**
- Beyond the scope of this course.

<CODE>

Delegate

<PLAY>