Protocol Extensions, Meet List Controllers

Lighter Controllers using protocol extensions

Jad Osseiran iOS/embedded engineer at Index

22 March 2016

Outline

What will we cover?

- Core concepts
  - What is a List Controller
  - Protocol extensions
- Previous abstraction technique
- Lighter controllers with protocol extensions

Core Concepts

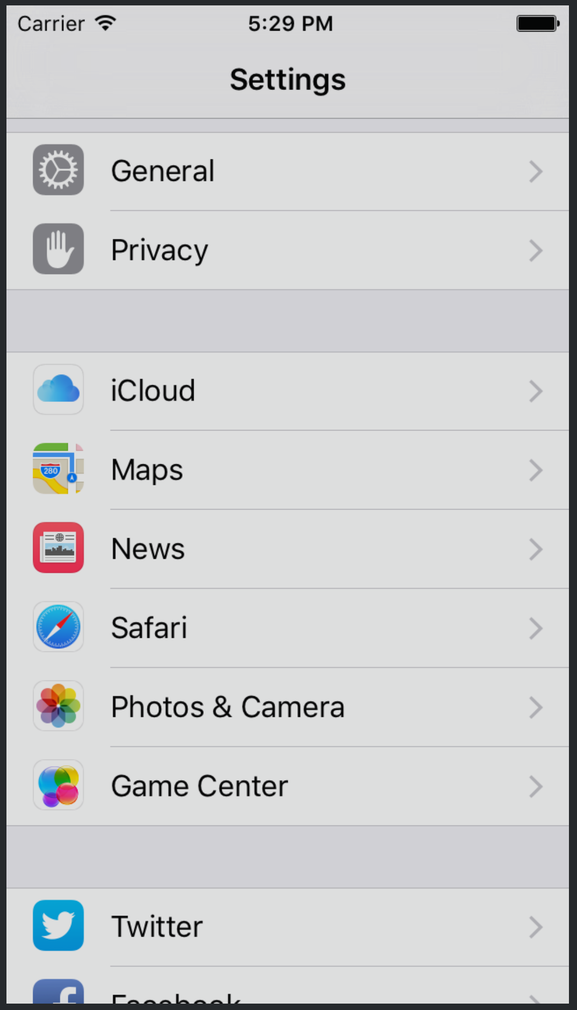
A little background before we delve in

List Controllers

`UITableViewController` & friends

- Tables, collections, custom lists
- Used all throughout iOS
- Delegate & data source methods
  - `tableView:didSelectRowAtIndexPath:`
  - `tableView:cellForRowAtIndexPath:`

They tend to get **bloated**



Protocol Extensions

Arguably the best Swift 2.0 feature

- Extend protocols to provide method and property implementations
- Extremely powerful

```
protocol StateMachine {
  mutating func reset()

  func next() -> Self?

  func previous() -> Self?
}
```

Protocol Extensions

Arguably the best Swift 2.0 feature

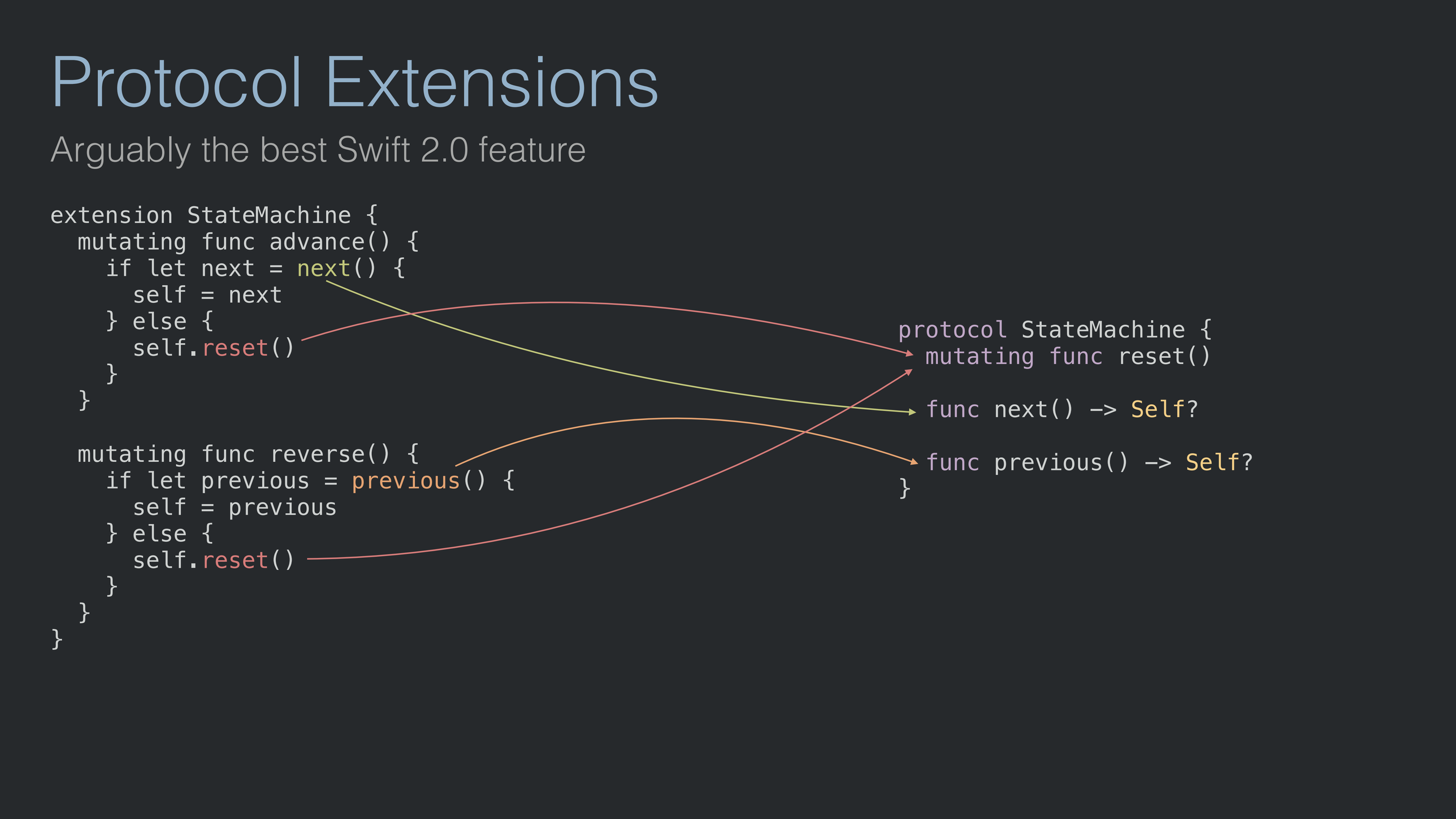
```
extension StateMachine {
  mutating func advance() {
    if let next = next() {
      self = next
    } else {
      self.reset()
    }
  }

  mutating func reverse() {
    if let previous = previous() {
      self = previous
    } else {
      self.reset()
    }
  }
}
```

Protocol Extensions

Arguably the best Swift 2.0 feature

```
extension StateMachine {
  mutating func advance() {
    if let next = next() {
      self = next
    } else {
      self.reset()
    }
  }

  mutating func reverse() {
    if let previous = previous() {
      self = previous
    } else {
      self.reset()
    }
  }
}
```

```
protocol StateMachine {
  mutating func reset()

  func next() -> Self?

  func previous() -> Self?
}
```



Previous Abstraction Technique

In the days prior to protocol extensions

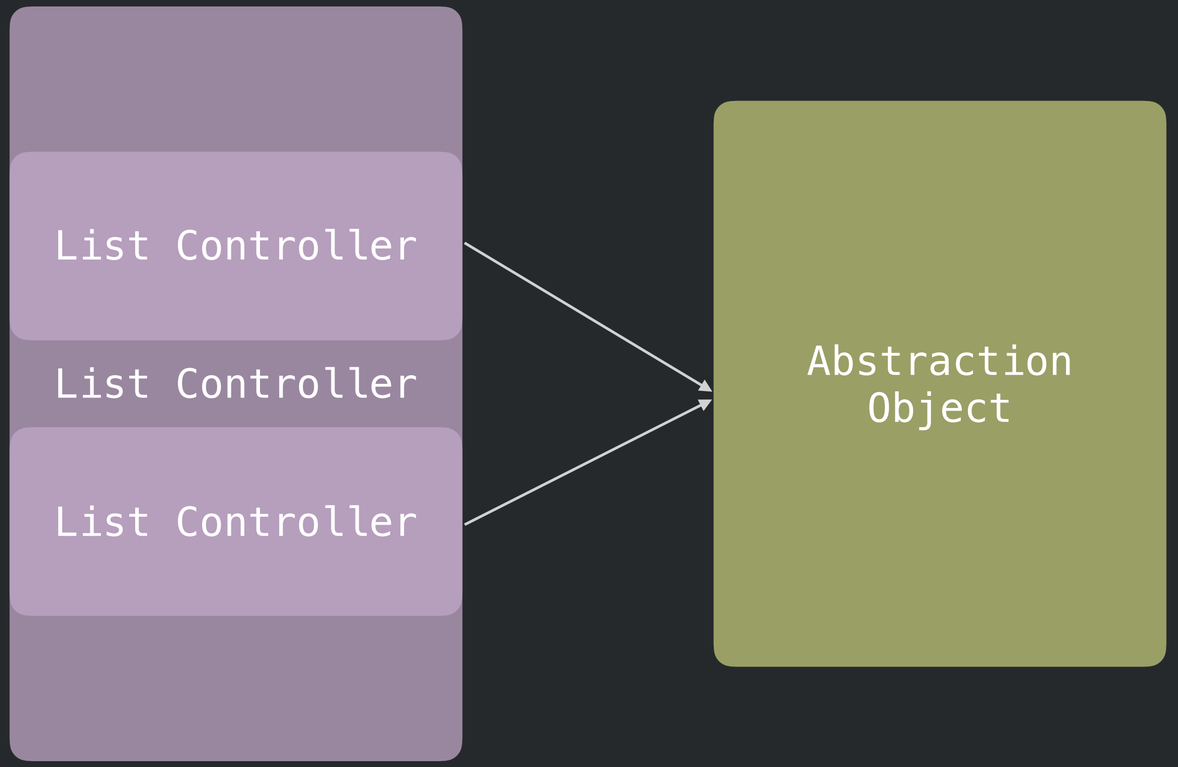
Abstraction Object

The brains of the list controller

- A class that conforms to the data source & delegate of the List Controller
- Abstracts the data logic code away
- In line with the MVC model

Abstraction Object

The brains of the list controller



Lighter Controllers With Protocol Extensions

Let’s extend some protocols

`List` Protocol

The master protocol

- The identification & configuration of a particular cell
- The behaviour of a particular cell selection

```
public protocol List {
  associatedtype ListView
  associatedtype Cell
  associatedtype Object

  func cellIdentifierForIndexPath(indexPath: NSIndexPath) -> String

  func listView(listView: ListView, configureCell cell: Cell,
                withObject object: Object,
                atIndexPath indexPath: NSIndexPath)

  func listView(listView: ListView, didSelectObject object: Object,
                atIndexPath indexPath: NSIndexPath)
}
```

`NonFetchedList` Protocol

Building for a simple static list

- Builds on `List`
- Provides basic behaviour for static list data

```
public protocol NonFetchedList: List {
  var listData: [[Object]]! { get set }
}

public extension NonFetchedList {
  var numberOfSections: Int { … }

  func numberOfRowsInSection(section: Int) -> Int { … }

  func objectAtIndexPath(indexPath: NSIndexPath) -> Object? { … }

  func isValidIndexPath(indexPath: NSIndexPath) -> Bool { … }
}
```

`TableList` Protocol

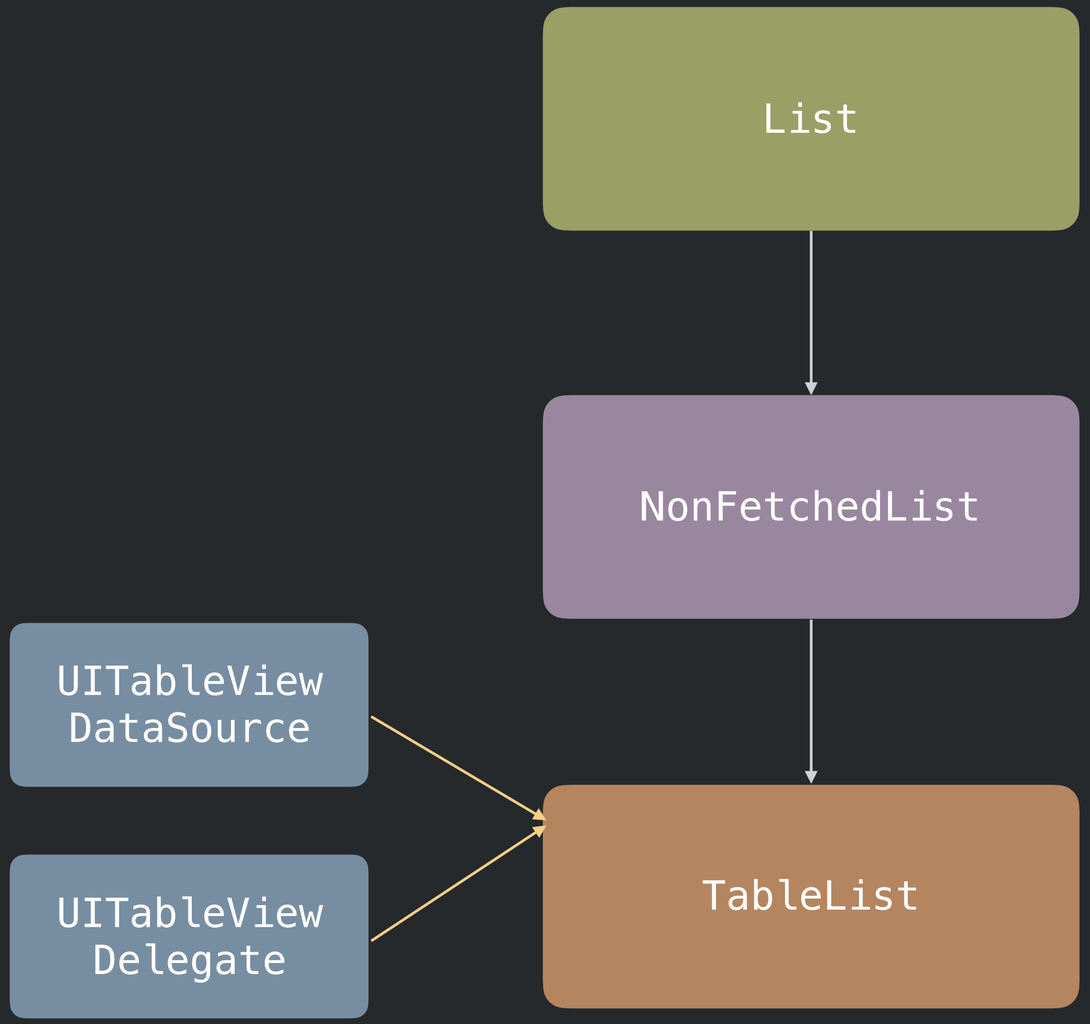
Specific List Controller: table

- Builds on `NonFetchedList`
- Conforms to `UITableViewDataSource` & `UITableViewDelegate`
- Sounds similar to our Abstraction Object

```
public protocol TableList: NonFetchedList, UITableViewDataSource, UITableViewDelegate {
  var tableView: UITableView! { get set }
}
```

`TableList` Protocol

Let's recap




```
extension NonFetchedList {
  numberOfSections
  numberOfRowsInSection:
  objectAtIndexPath:
  isValidIndexPath:
}
```

```
extension TableList {
  tableCellAtIndexPath:
  tableDidSelectItemAtIndexPath:
}
```

`TableList` Protocol Extension

Leveraging protocol extension

```
public extension TableList where ListView == UITableView, Cell == UITableViewCell {
  func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell {
    let identifier = cellIdentifierForIndexPath(indexPath)
    let cell = tableView.dequeueReusableCellWithIdentifier(identifier,
                                                           forIndexPath: indexPath)

    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, configureCell: cell, withObject: object, atIndexPath: indexPath)
    }

    return cell
  }

  func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) {
    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, didSelectObject: object, atIndexPath: indexPath)
    }
  }
}
```

`TableList` Protocol Extension

More specific, less generic

```
public extension TableList where ListView == UITableView, Cell == UITableViewCell {
  func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell {
    let identifier = cellIdentifierForIndexPath(indexPath)
    let cell = tableView.dequeueReusableCellWithIdentifier(identifier,
                                                           forIndexPath: indexPath)

    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, configureCell: cell, withObject: object, atIndexPath: indexPath)
    }

    return cell
  }

  func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) {
    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, didSelectObject: object, atIndexPath: indexPath)
    }
  }
}
```

`TableList` Protocol Extension

From the delegate & data source of `UITableView`

```
public extension TableList where ListView == UITableView, Cell == UITableViewCell {
  func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell {
    let identifier = cellIdentifierForIndexPath(indexPath)
    let cell = tableView.dequeueReusableCellWithIdentifier(identifier,
                                                           forIndexPath: indexPath)

    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, configureCell: cell, withObject: object, atIndexPath: indexPath)
    }

    return cell
  }

  func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) {
    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, didSelectObject: object, atIndexPath: indexPath)
    }
  }
}
```

`TableList` Protocol Extension

From `NonFetchedList`

```
public extension TableList where ListView == UITableView, Cell == UITableViewCell {
  func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell {
    let identifier = cellIdentifierForIndexPath(indexPath)
    let cell = tableView.dequeueReusableCellWithIdentifier(identifier,
                                                           forIndexPath: indexPath)

    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, configureCell: cell, withObject: object, atIndexPath: indexPath)
    }

    return cell
  }

  func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) {
    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, didSelectObject: object, atIndexPath: indexPath)
    }
  }
}
```

`TableList` Protocol Extension

From `List`

```
public extension TableList where ListView == UITableView, Cell == UITableViewCell {
  func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell {
    let identifier = cellIdentifierForIndexPath(indexPath)
    let cell = tableView.dequeueReusableCellWithIdentifier(identifier,
                                                           forIndexPath: indexPath)

    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, configureCell: cell, withObject: object, atIndexPath: indexPath)
    }

    return cell
  }

  func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) {
    if let object = objectAtIndexPath(indexPath) {
      listView(tableView, didSelectObject: object, atIndexPath: indexPath)
    }
  }
}
```

Can we do better?

Of course we can

`FetchedList` Protocol

Building for a Core Data dynamic list

- Behaviour for a `NSFetchedResults` backed `List`
- Allows for easy cell updating and dynamic lists

```
public protocol FetchedList: List, NSFetchedResultsControllerDelegate {
  var fetchedResultsController: NSFetchedResultsController! { get set }
}

public extension FetchedList {
  var numberOfSections: Int { … }
  var sectionIndexTitles: [AnyObject]? { … }

  func numberOfRowsInSection(section: Int) -> Int { … }
  func isValidIndexPath(indexPath: NSIndexPath) -> Bool { … }
  func objectAtIndexPath(indexPath: NSIndexPath) -> AnyObject? { … }
  func titleForHeaderInSection(section: Int) -> String? { … }
}
```

FetchedTableList Protocol

NSFetchedResults backed table

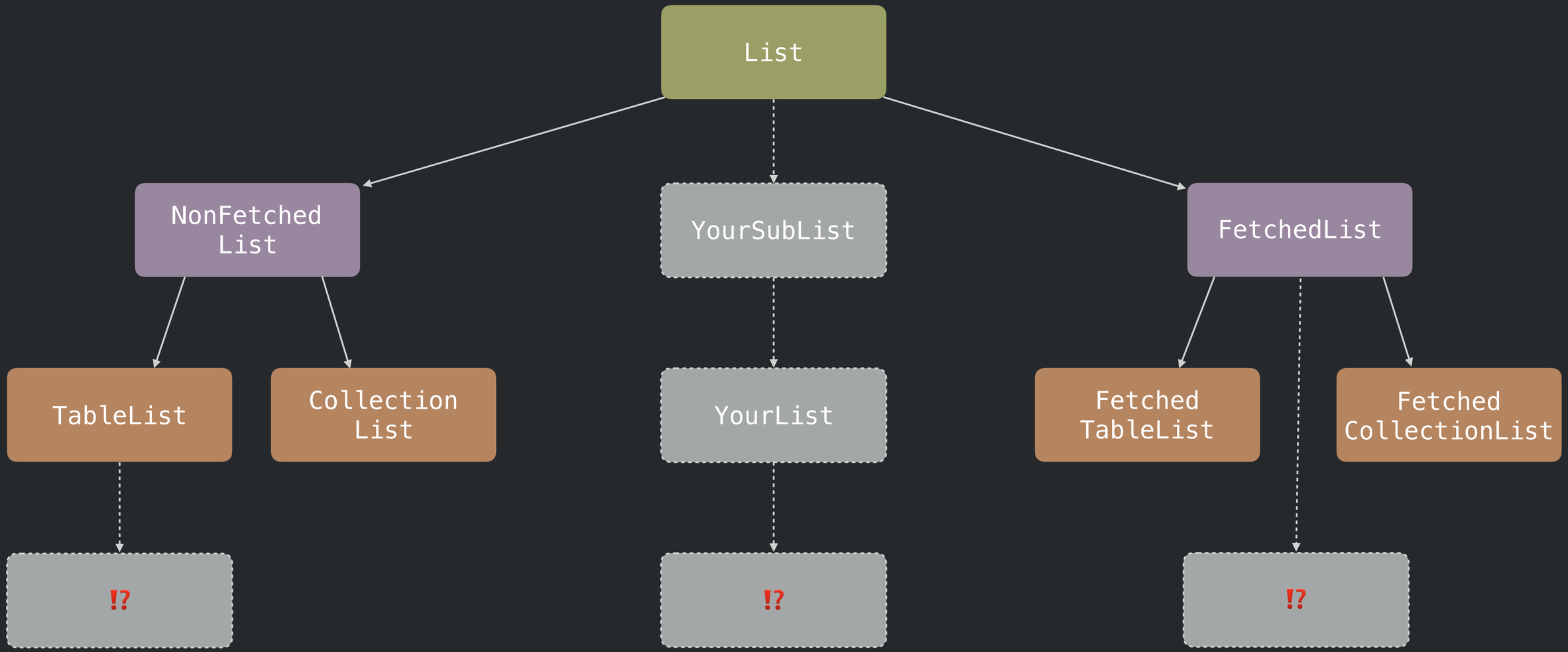
```
public protocol FetchedTableList: FetchedList, UITableViewDataSource, UITableViewDelegate {
    var tableView: UITableView! { get set }
}

public extension FetchedTableList where ListView == UITableView, Cell == UITableViewCell,
  Object == AnyObject {
    func tableCellAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell { … }
    func tableDidSelectItemAtIndexPath(indexPath: NSIndexPath) { … }
}

public extension FetchedTableList where ListView == UITableView, Cell == UITableViewCell,
  Object == AnyObject {
    func tableWillChangeContent() { … }
    func tableDidChangeSection(sectionIndex: Int,
        withChangeType type: NSFetchedResultsChangeType) { … }
    func tableDidChangeObjectAtIndexPath(indexPath: NSIndexPath?,
                                         withChangeType type: NSFetchedResultsChangeType,
                                         newIndexPath: NSIndexPath?) { … }
    func tableDidChangeContent() { … }
}
```

Summary

We’re nearly done

Approach Overview

A picture is worth a thousand words



Pain Points

Ouch

- Method dispatching with protocol extensions
  - `tableCellAtIndexPath:` vs. `tableView:cellForRowAtIndexPath:`
  - Deserves a talk in its own right
- Not Objective-C compatible

More Information

Where to go from here

Open source repo:
https://github.com/jad6/JadKit

@SwiftLang repo:
https://github.com/apple/swift

Protocol extensions & method dispatching
http://nomothetis.svbtle.com/the-ghost-of-swift-bugs-future

```
questions?.
filter({jad.canAnswer($0)})
```