Be Nice to Your Future Self

Writing Code to Optimize Maintainability

Premises

- Code is written a few times, but read many times
- All code is optimized, even if it's a default optimization
- You could be your own worst enemy (or at least be self destructive)

Problem Statement

Features are conceived to solve a current need, and they are constrained by UI/UX, release dates, and performance metrics.

Features can't be conceived to solve unknown future needs, nor are they constrained by maintainability.

Outcome

- Unreadable Code
  - Magic values or data adjustments from unknown requirements
  - Hacks on hacks to glue the different requirements together
- Multiple requirements mixed into a single function
  - If-else for each requirement
  - Plain old spaghetti code
- Untested Code
  - Edge cases from mixed requirements: unexpected code paths

Outcome (Best Case)

- Codebase is a complex mix of requirements
  - Different edge cases can conflict
  - Mixing of similar but not identical business logic
- Dead code and unit tests
  - Full features that cannot be accessed
  - Complex handling for truly impossible conditions

Path to a Better Outcome

- Extending APIs to Build Your App
- Data Source, Business Logic, Model, View, Controller
- Writing Code for Humans

Extending APIs

Balancing Top Down and Bottom Up Development

What is Top Down Development (to me)?

- Creating custom classes and structs that define a problem domain
    - Models are created to represent data
    - Methods in controllers or models transform the data based on user interaction and business logic
- The default way of developing an app

Simple Example

```
struct ChecklistItem { // Model
    var title: String
    var checked: Bool
}

class CheckedChecklistViewController : UITableViewController {
    var checklist = [ChecklistItem(title: "test", checked: true)]

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        var count = 0
        for checklistItem in checklist {
            if checklistItem.checked {
                count = count + 1
            }
        }
        return count
    }
}
```

What is Bottom Up Development?

- Extending the language, core APIs, and third-party APIs

Simple Example

```swift
extension Collection where Iterator.Element == ChecklistItem {
    func countOfChecked() -> Int {
        return self.reduce(0) { result, checklistItem in
            result + (checklistItem.checked ? 1 : 0)
        }
    }
}

struct ChecklistItem { // Model
    var title: String
    var checked: Bool
}

class CheckedChecklistViewController : UITableViewController {
    var checklist = [ChecklistItem(title: "test", checked: true)]

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return checklist.countOfChecked()
    }
}
```

What's the Difference?

- Clearer meaning
  - The number of checked items is an attribute of the array
- Separation from unrelated classes
  - Keep view controllers focused on binding data to view and handing user input
  - Keep models away from processing data

What to Extend

- Find the noun
    - Talk the task out in a sentence: the noun of the sentence should be class to extend
    - If there is more than one noun, use a static method of the return value

Where to Extend

- Local file
  - If the extension is only needed in one source file, there may be no reason to create a new file for it
- Extensions file
  - For simple extensions used in multiple places, pool them together into an Extensions.swift file
- Class extension file
  - For a group of extensions or a complex extension used in multiple places, use a specific file for them

Keys to Bottom Up

- Balance balance balance
  - Look for the best, most natural fit
- Start local and group when needed
  - Let your future self determine the best way to group functionality

Data Source

An old idea based on new experience

Method Soup

- Fetching data from many similar Web service endpoints with different numbers of parameters
- Different classes provided data to different fetch method
- There were more than ten Web service endpoints
- There were as many a five methods for each Web service endpoint

Simple Example

```
func requestEngines(year: String, make: String, model: String,
bodyStyle: String, driveType: String) -> EngineDataSetPromise

func requestEngines(year: String, make: String, model: String,
bodyStyle: String, cabBedStyle: String, driveType: String,
towCapacity: Int) -> EngineDataSetPromise

func requestEngines(year: String, make: String, modelCode: String,
driveType: String) -> EngineDataSetPromise

func requestEngines(year: String, make: String, modelCode: String,
bodyStyle: String, cabBedStyle: String, driveType: String,
towCapacity: Int) -> EngineDataSetPromise

func requestBodyStyles(year: String, make: String, model: String,
engine: String, driveType: String) -> BodyStyleDataSetPromise

func requestBodyStyles(year: String, make: String, model: String,
engine: String, driveType: String, cabBedStyle: String) ->
BodyStyleDataSetPromise
```

<Model>Fetchable Protocol

- Each Web service endpoint has a fetchable protocol
- Fetch methods have two parameters: a object conforming to the correct fetchable protocol, and a completion block returning the data
- Different classes used as input to a fetch just need to conform to fetchable

Simple Example

```swift
func fetchEngines(_ fetchable: EngineFetchable) ->
EngineDataSetPromise

func fetchBodyStyles(_ fetchable: BodyStyleFetchable) ->
BodyStyleDataSetPromise
```

<Model>DataSource Class

- Groups related fetch methods, usually has no properties or state
- Create on-the-fly in multiple places

Simple Example

```
class VehicleDataSource {

    func fetchEngines(_ fetchable: EngineFetchable) ->
EngineDataSetPromise {
        /* … */
    }

    func fetchBodyStyles(_ fetchable: BodyStyleFetchable) ->
BodyStyleDataSetPromise {
        /* … */
    }

}
```

Backdoor to an Old Idea

- Key Attributes of DataSource
  - Lightweight
  - Implementation agnostic API
  - Data set result
- Similar to a Data Access Object

Simple Example

```
class ChecklistDataSource {

    func create() -> ChecklistDataSetPromise {
        /* … */
    }

    func fetch(_ fetchable: ChecklistFetchable) -> ChecklistDataSetPromise {
        /* … */
    }

    func update(dataSet: ChecklistDataSet) -> ChecklistDataSetPromise {
        /* … */
    }

    func delete(dataSet: ChecklistDataSet) -> ChecklistDataSetPromise {
        /* … */
    }

}
```

Business Logic

The hard work of the hard work

Business Rules Hidden Everywhere

- In most apps, the business logic is spread out between the models and the view controllers
- It can be hard to determine all the business rules in place to perform a business transaction without debugging the code

Business Transactions

- Create methods to execute business transactions
    - Contains the logic in one place
- One business transaction per method
    - Ensures requirements and business rules don't mix

Simple Example

```
func loadAllChecklists(into checklists: ChecklistDataSet)

func insertNewChecklist(title: String, into checklists: ChecklistDataSet, at index: Int)

func deleteChecklist(from checklists: ChecklistDataSet, at index: Int)

func renameChecklist(title: String, in checklists: ChecklistDataSet, at index: Int)
```

<Model>BusinessLogic Class

- Manages the business rules needed to complete a business transaction
- Each business transaction has a single method to complete the transaction
- Can be a stateless lightweight object, but is more likely to be a Singleton

Simple Example

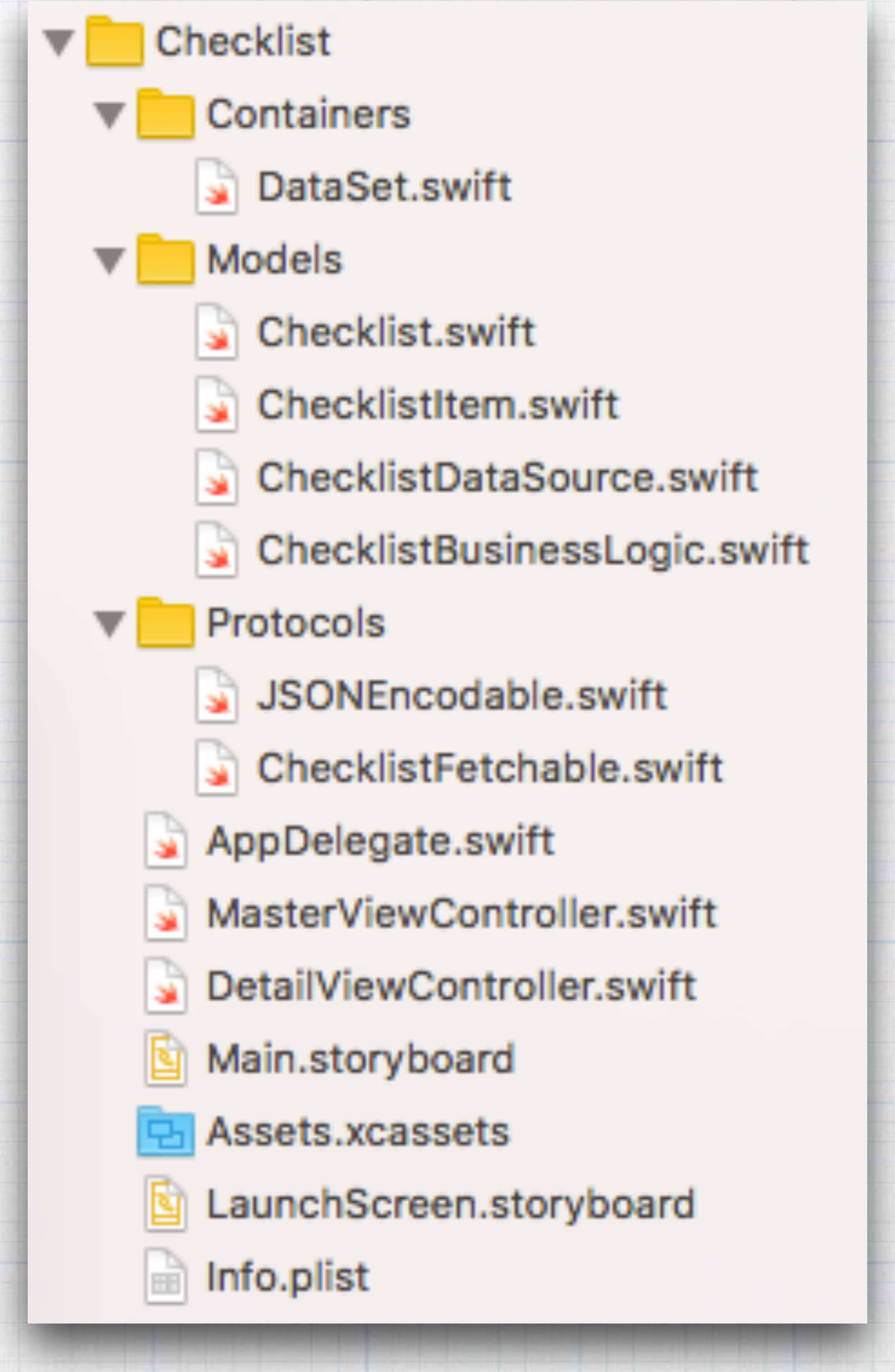
```swift
class ChecklistBusinessLogic {

    static let sharedInstance = ChecklistBusinessLogic()

    func loadAllChecklists(into checklists: ChecklistDataSet) {
        /* … */
    }

    func insertNewChecklist(title: String, into checklists: ChecklistDataSet, at index: Int) {
        /* … */
    }

    func deleteChecklist(from checklists: ChecklistDataSet, at index: Int) {
        /* … */
    }

    func renameChecklist(title: String, in checklists: ChecklistDataSet, at index: Int) {
        /* … */
    }

}
```

Writing Code for Humans

Make Use Cases Obvious

Yes, but What Does It Do?

If you're a new developer assigned to work on the app, how could you tell what the app does, or how it does it?



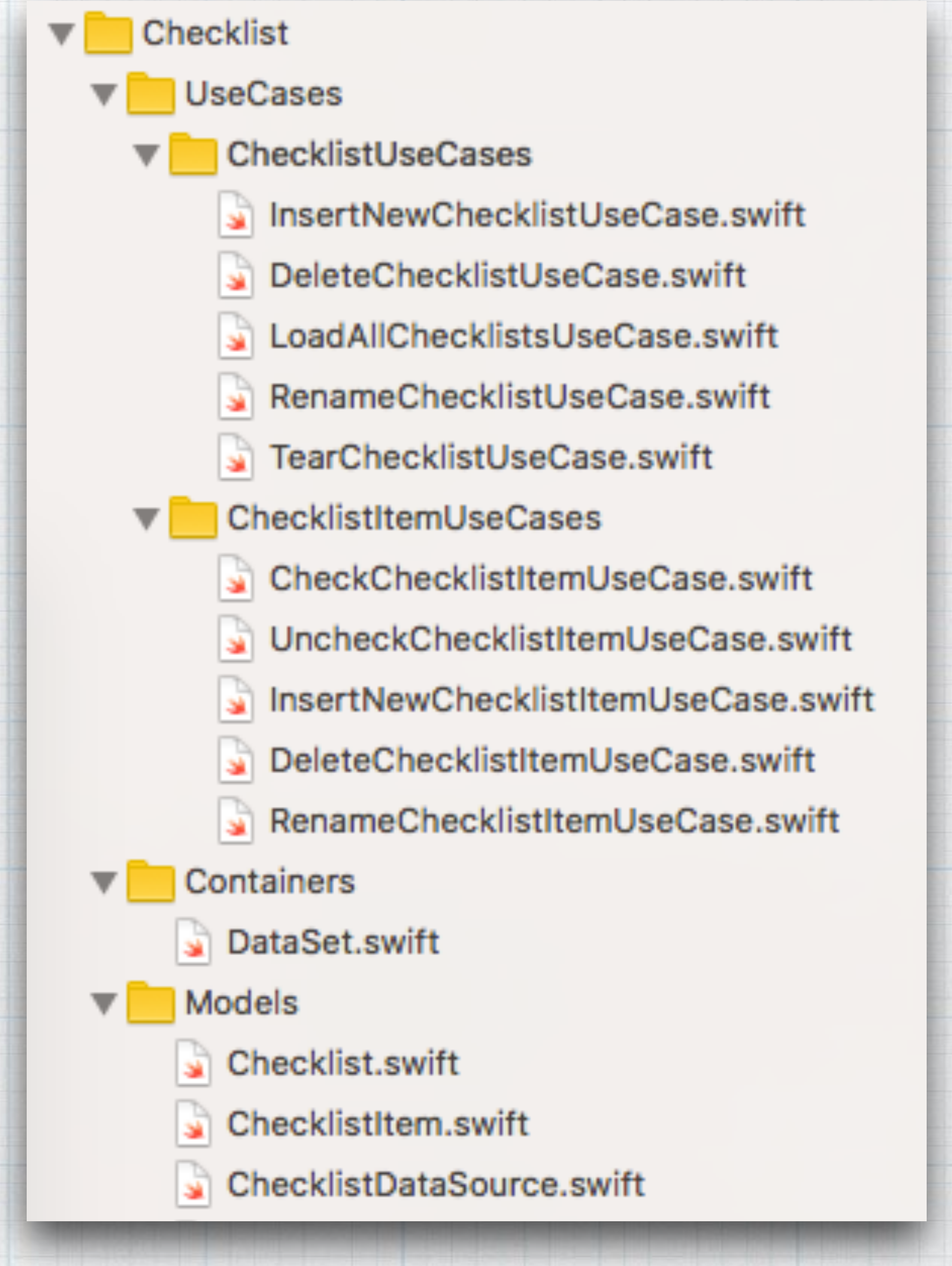
```
▼ Checklist
    ▼ Containers
        DataSet.swift
    ▼ Models
        Checklist.swift
        ChecklistItem.swift
        ChecklistDataSource.swift
        ChecklistBusinessLogic.swift
    ▼ Protocols
        JSONEncodable.swift
        ChecklistFetchable.swift
    AppDelegate.swift
    MasterViewController.swift
    DetailViewController.swift
    Main.storyboard
    Assets.xcassets
    LaunchScreen.storyboard
    Info.plist
```

Writing Code for Humans

- Create use cases as top level artifacts
- Layout the files to show the domain of your application

Now, What Does It Do?

As a new developer I can see what the app does. I can make guess as to where to look when debugging



- Checklist
  - UseCases
    - ChecklistUseCases
      - InsertNewChecklistUseCase.swift
      - DeleteChecklistUseCase.swift
      - LoadAllChecklistsUseCase.swift
      - RenameChecklistUseCase.swift
      - TearChecklistUseCase.swift
    - ChecklistItemUseCases
      - CheckChecklistItemUseCase.swift
      - UncheckChecklistItemUseCase.swift
      - InsertNewChecklistItemUseCase.swift
      - DeleteChecklistItemUseCase.swift
      - RenameChecklistItemUseCase.swift
  - Containers
    - DataSet.swift
  - Models
    - Checklist.swift
    - ChecklistItem.swift
    - ChecklistDataSource.swift

Bringing It Together

Combining extensions, data source, and Writing Code for Humans

Aligning Terminology

- Business transaction are use cases
  - Use cases should be business transactions
  - Each business transaction should have a use case

The Mechanics

- Create a near empty business logic class
    - Keep only the needed properties
- Use extensions to the business logic class, model classes, and framework classes
- Keep all the single use extensions in the use case file
- Shared extensions are placed in separate common files grouped by functionality or class

Simple Example

```
extension ChecklistBusinessLogic {
    func insertNewChecklist(title: String, at index: Int) -> Promise<Void> {
        let checklistsRaceConditionSafe = self.checklists
        return firstly {
            self.dataSource.create()
        } .then { dataSet in
            Checklist(id: dataSet.items[0].id, title: title, items: dataSet.items[0].items)
        } .then { checklist in
            self.dataSource.update(dataSet: ChecklistDataSet(items: [checklist]))
        } .then { dataSet in
            self.checklists = checklistsRaceConditionSafe.inserted(dataSet.items[0], at: index)
        }
    }
}
extension Array {
    func inserted(_ element: Element, at index: Int) -> Array {
        var result = self
        result.insert(element, at: index)
        return result
    }
}
```

New Outcome

The promise of a maintainable future

Readable Code

- The code in the Business Logic extensions focuses on one use case at a time

Distinct Business Logic

- Separate files for different use cases helps prevent requirement mixing

Obsolete Use Cases are Easier To Remove

- Start with removing the use case file and see what else is effected

Final Thoughts

- Write your own code for your own future self
- Pay attention to where you look for code
  - Put your code there
- Write clean code over clever code
  - Your future self will not be impressed
- Keep coupling as low a possible
- Short methods and small files

Writing Code for Humans, Not Computers

http://www.slideshare.net/renecacheaux/writing-code-for-humans-not-computers
http://360idev.com/session-videos/
https://vimeopro.com/360conferences/360idev-2016/video/182462162

Screaming Architecture

https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html

Architecting iOS Apps with VIPER

https://www.objc.io/issues/13-architecture/viper/

Jeffery Thomas

jeffery.thomas@gmail.com
@jeffery_thomas