CS 326
Dependencies and Decoupling

Stephen Freund

Limits of Scaling

• What prevents us from building huge, intricate structures that work perfectly and indefinitely?
  – Not just friction
  – Not just gravity
  – Not just wear-and-tear
• ... The difficulty of managing complexity
• Modularity, and minimize interactions

Cohesion and Coupling

• Split Design into parts that don't interact much
  – Coupling: amount of interaction between parts
  – Cohesion: similarity/behavior within a part

Design Exercise #1

• Write a typing-break reminder program
  – Offer user occasional reminders to take a break from typing a stretch.

  • Naive Design:
    – Main program makes a timer
    – Timer loops performs actions periodically
    – Action = "Display message and offer exercises"
    – (Ignore fancy multi-threaded solutions...)
Code, Version 1

```swift
// MARK: Module 1: Time To Stretch
class TimeToStretch {
    public func run() {
        print("Take a Break!
             \(suggestExercise())")
    }

    private func suggestExercise() -> String {
        if Int(arc4random_uniform(2)) == 0 {
            return "Touch Toes"
        } else {
            return "Do 100 Push Ups"
        }
    }
}

// MARK: Module 2: Timer
class Timer {
    private let tts = TimeToStretch()

    public func start() {
        while true {
            sleep(3)
            tts.run()
        }
    }
}

// MARK: Module 3: Main
let timer = Timer()
timer.start()
```

Module Dependency Diagram (MDD)

- Arrow indicates “depends on” or “knows about”
  - Simplest notion: any name mentioned in the source code

- What’s wrong with this diagram?
  - Does Timer really need to depend on TimeToStretch?
  - Is Timer re-usable in a new context?

Decoupling

- Timer needs to call the run method
  - but doesn’t need to know what exactly it does
- Weaken dependency of Timer on TimeToStretch
  - Introduce a weaker specification, in the form of an interface or abstract class

```
public protocol TimerTask {
    func run()
}
```

- Timer works with anything conforming to TimerTask

Code, Version 2

```swift
// MARK: Module 2: Timer
class Timer {
    private let task: TimerTask

    public init(task: TimerTask) {
        self.task = task
    }

    public func start() {
        while true {
            sleep(3)
            task.run()
        }
    }
}

// MARK: Module 1: Time To Stretch
class TimeToStretch: TimerTask {
    public func run() {
        print("Take a Break!
             \(suggestExercise())")
    }

    private func suggestExercise() -> String {
        ...
    }
}

// MARK: Module 3: Main
let timer = Timer(TimeToStretch())
timer.start()
```
MDD (version 2)

- Timer depends only on TimerTask
  - Timer is much easier to reuse
  - Main depends on the constructor for TimeToStretch
- Main still depends on Timer (is this necessary?)

Callback Design Pattern

- Synchronous Callback
- Time increases down
- Solid Arrow: Call
- Dotted Arrow: Return
- Call Stack:
- Benefit: Library does not depend on my service, only on some super type of my service

Synchronous Callback Examples

- Collections:
  - Array, Dictionary, Set
  - map/filter/first/etc.
- UIKit Delegates
  - UITableViewDataSource
- Useful when callback result is immediately needed

Synchronous Callback Examples

- Collections:
  - Array, Dictionary, Set
  - map/filter/first/etc.
- UIKit Delegates
  - UITableViewDataSource
- Useful when callback result is immediately needed
Asynchronous Callback Examples

- GUI listeners
  - Register interest in event and where to call back (@IBAction)
  - Useful when callback should run when something interesting occurs
- Tasks to put on DispatchQueues

Code, Version 3

```swift
public protocol TimerTask {
    func run()
}

// MARK: Module 1: Time To Stretch
public class TimeToStretch : TimerTask {
    private var timer : Timer?

    public func start() {
        timer = Timer(self)
        timer?.start()
    }

    public func run() {
        print("Take a Break!\n\n\n\nsuggestExercise()")
    }

    private func suggestExercise() -> String {
        ...
    }
}

// MARK: Module 2: Timer
public class Timer {
    private let task : TimerTask

    public init(_ task : TimerTask) {
        self.task = task
    }

    public func start() {
        while true {
            sleep(3)
            task.run()
        }
    }
}

// MARK: Module 3: Main
let tts = TimeToStretch()
tts.start()
```

Closures vs. Objects w/ Protocol

- Closures are great for "one-off" callbacks
  - Example: Sort an array
    - `array.sort(by: { $0.dst < $1.dst })`
  - Decouples `sort()` from the comparison
- Object w/ Protocols better for multiple, related closures
  - Eg: `Comparable, TableViewDataSource`
  - Common terminology for concept, can store data
  - Provides type and organization for idiom
**Decoupling and Design**

- A good design has dependences (coupling) only where it makes sense
- During design, examine dependences
  - **Before you code**
  - Don’t introduce unnecessary coupling
- Coupling is an easy temptation if you code first
  - You realize a method needs information from another object and hack in a way to get that information:
    - It might be easy to write
    - It will damage the code’s modularity and reusability

**Design Exercise #2**

- A program to display information about stocks
  - Stock tickers
  - Spreadsheets
  - Graphs

- Naive Design
  - Make a class to represent stock information
  - That class updates all views of that information when it changes
    - graphs, tickers, portfolio networth, etc.

**The Old Way...**

```swift
class Stocks {
    
    func updateViewers() {
        ticker.update(priceInfo)
        spreadsheet.update(priceInfo)
        graph.update(priceInfo)
    }
    
    // Main
    let stocks = new Stocks()
    
    stocks.updateViewers()
}
```

**MDD**

- **Problem**
  - To add/change a viewer, must change **Stocks**
- **Better Design**
  - insulate **Stocks** from the vagaries of the viewers
The Observer Pattern

- Stocks keeps list of PriceObservers, notifies them of changes
- Main: creates viewers and passes them to Stocks as observers

Weaken the Coupling

protocol PriceObserver {
    func update(priceInfo: PriceInfo)
}

class Stocks {
    private var observers = [PriceObserver]()

    public func add(observer: PriceObserver) {
        observers.append(observer)
    }

    private func notify(priceInfo: PriceInfo) {
        for observer in observers {
            observer.update(priceInfo: priceInfo)
        }
    }
}

Push vs. Pull

- Observer Pattern implements push functionality.
- Pull model: give viewers access to Stocks, let them extract the data they need.

“Push” versus “Pull” efficiency can depend on frequency of operations
  - Also possible to use both patterns simultaneously.
class Stocks {
    ...
    func info() : ... { ... }
}

class Chart {
    let stock : Stocks
    ...
}

// Main:
let stocks = Stocks()
let chart = Chart(stocks)
let ticker = Ticker(stocks)
let sheet = Spreadsheet(stocks)

protocol Observer {
    func update(info : Observable)
}

class Observable {
    var observers = [Observer]()
    func add(observer : Observer) {
        observers.append(observer)
    }
    func notify() {
        for observer in observers {
            observer.update(info: self)
        }
    }
}

class SignupSheet : Observable {
    var students = [String]()
    func add(student : String) {
        students.append(student)
        notify()
    }
}

class SignupWatcher : Observer {
    func update(info: Observable) {
        if let sheet = info as? SignupSheet {
            print("Count is now \(sheet.students.count)")
        }
    }
}
**UIs: MVC to Limit Dependencies**

- Avoid tangling data, logic, and appearance
- Easy for dependencies between model/controller/view to creep in
  - It’s happened to all of us...
  - Leads to complex, brittle, non-reusable code
- Callbacks, delegates, listeners, observers reduce coupling
  - Central in UIKit and other frameworks

**Shared Constraints**

- Coupling can result from “shared constraints” from specs, not just code dependencies
  - If one fails to write the correct format, the other will fail to read
- Shared constraints are easier to reason about if they are well encapsulated
  - A single module should contain and hide all information about the format

**Facade Design Pattern**

- Want to support secure file copying to a server
  - You have a powerful general purpose library
  - But a secure file copy exposes its complexity
  - Creates many dependencies on library components
Facade Design Pattern

- Build a interface to that library to hide the (mostly irrelevant) complexity