

Many Types Make Light Work

@rob_rix 💖 rob.rix@github.com

Reusing code

- change happens: bugs, features, platform changes
- complexity scales with quantity of code
- programming is managing complexity under change
- reusing code is necessary (if insufficient)
- reusing *code* means
 - reusing *implementations*
 - reusing *interfaces*

Reusing implementations

- Don’t Repeat Yourself (DRY)
- functions, methods, (sub)classes, &c.

Reusing interfaces

- lets the same code work with different types
- subclassing, protocols

Subclassing

- inherit superclass' interface & implementation
- describes the class hierarchy at compile time

Composition

- composition is when you put a thing in a thing and then it's a thing then using objects together
- describes the object graph at runtime

The trouble with subclassing

- it *conflates* reusing interfaces with reusing implementations
- it *couples* subclasses to superclass implementations
- it *encourages* tight coupling in composed classes



Don't subclass.

Approach 1:

Factor class hierarchies out

Encapsulate the concept that
varies.

— Gamma, Helm, Johnson, & Vlissides' *Design Patterns*

Factor out independent code

```
class Post {  
    let title: String  
    let author: String  
    let postedAt: NSDate  
    let URL: NSURL  
}  
  
class Tweet: Post { ... }  
class RSS1Post: XMLPost { ... }  
class RSS2Post: XMLPost { ... }  
class AtomPost: XMLPost { ... }  
  
class XMLPost: Post {  
    let XMLData: NSData  
    let titlePath: XPath  
    let authorPath: XPath  
    let postedAtPath: XPath  
    let URLPath: XPath  
}
```

Factoring out independent code

- tightly couples model classes to parsing strategies
- needless margin for error (e.g. initializing abstract classes)

Factoring out independent code

```
class Post {  
    let title: String  
    let author: String  
    let postedAt: NSDate  
    let URL: NSURL  
}  
  
class Tweet: Post { ... }  
class RSS1Post: Post { ... }  
class RSS2Post: Post { ... }  
class AtomPost: Post {  
    init(data: NSData) {  
        let parser = XMLParser(data)  
        super.init(title: parser.evaluateXPath(...), ...)  
    }  
}  
  
class XMLParser { ... }
```

Approach 2:

Protocols, not superclasses

Protocols are interfaces

- specify required properties & methods
- Cocoa uses protocols for several purposes
 - delegate/data source protocols (e.g. `UITableViewDelegate`)
 - model protocols (e.g. `NSDraggingInfo`, `NSFetchedResultsSectionInfo`, `NSFilePresenter`)
 - behaviour protocols (e.g. `NSCoding`, `NSCopying`)

Protocols are shared interfaces

```
class Post {  
    let title: String  
    let author: String  
    let postedAt: NSDate  
    let URL: NSURL  
}  
  
class Tweet: Post { ... }  
class RSS1Post: Post { ... }  
class RSS2Post: Post { ... }  
class AtomPost: Post {  
    init(data: NSData) {  
        let parser = XMLParser(data)  
        super.init(title: parser.evaluateXPath(...), ...)  
    }  
}
```

Using protocols to share interfaces

```
protocol Post {
    var title: String { get }
    var author: String { get }
    var postedAt: NSDate { get }
    var URL: NSURL { get }
}

class AtomPost: Post {
    let title: String
    let author: String
    let postedAt: NSDate
    let URL: NSURL

    init(data: NSData) {
        let parser = XMLParser(data)
        title = parser.evaluateXPath(...)
        ...
    }
}
```

Factor your protocols, too

- UITableViewDelegate API ref is in *9 sections*
- UITableViewDataSource & UITableViewDelegate aren't *really* independent
- exact same problem as with ill-factored classes
- factor around independent concepts instead

Delegate protocols suggest better factoring

- they force a single class to handle disparate concerns
- instead, “encapsulate the concept that varies”
 - factor out view elements (e.g. rows) instead of delegate methods for contents/behaviour
 - KVO-compliant selected/displayed subset properties (or signals) instead of will/did delegate callbacks
 - can start by splitting methods into tiny protocols

Approach 3:

Minimize interfaces with
functions

Function overloading is almost an interface

- `first(...)` returns the first element of a stream/list

```
func first<T>(stream: Stream<T>) -> T? { ... }
func first<T>(list: List<T>) -> T? { ... }
```

- `dropFirst(...)` returns the rest of a stream/list following the first element

```
func dropFirst<T>(stream: Stream<T>) -> Stream<T> { ... }
func dropFirst<T>(list: List<T>) -> List<T> { ... }
```

Function overloading is not really an interface

- `second(...)` returns the second item in a list or stream
- But we can't write `second(...)` generically without a real interface

```
func second<T>(...?!) -> T? { ... }
```

Generic functions over protocols

```
protocol ListType {  
    typealias Element  
    func first() -> Element?  
    func dropFirst() -> Self  
}  
  
func second<L: ListType>(list: L) -> Element? {  
    return list.dropFirst().first()  
}
```

Function types are shared interfaces

```
struct GeneratorOf<T> : GeneratorType {  
    init(_ nextElement: () -> T?)  
  
    // A convenience to wrap another GeneratorType  
    init<G : GeneratorType where T == T>(_ base: G)  
  
    ...  
}
```

Approach 4:

Abstract into (many)
minimal types

Lists as a protocol

```
protocol ListType {  
    typealias Element  
    init(first: Element, rest: Self?)  
    var first: Element { get }  
    var rest: Self?  
}
```

How many different implementations of lists do we need, exactly?

Lists as a minimal type

```
enum List<T> {
    case Cons(Box<T>, Box<List<T>>)
    case Nil(Box<T>)

    var first: T { ... }
    var rest: List<T>? { ... }
}
```

fin

enums are fixed shared interfaces

Use enum for fixed sets of alternatives:

```
enum Result<T> {  
    case Success(Box<T>)  
    case Failure(NSError)  
}
```

Caveat: Cocoa *requires* you to subclass

Write minimal subclasses

- can you configure an instance instead of subclassing?
- extract distinct responsibilities into their own types
- code defensively

A final piece of advice

- make all classes final by default
- only remove final as a conscious choice
- consider leaving a comment as to why you did

Takeaway

- subclassing is for the weak and timid
- reuse interfaces with protocols
- reuse implementations by factoring & composing

Thanks to

Matt Diephouse, Ken Ferry, Kris
Markel, Andy Matuschak, Ryan
McCuaig, Kelly Rix, Justin Spahr-
Summers, Patrick Thomson...

...and you ❤️