Many Types Make Light Work[1]

🐦 @rob_rix
🐙 @robrix
📩 rob.rix@github.com

[1] https://github.com/robrix/Many-Types-Make-Light-Work

More code = more complexity = more bugs

Code reuse reduces risk.

- implementations:
  - Don’t Repeat Yourself (DRY)
  - functions, methods, (sub)classes, &c.
- interfaces:
  - use the same code 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*~~ combining code
— describes the runtime object graph/call stack

subclassing ≃ 💥🔥💀

Subclassing *conflates* interface & implementation reuse

Subclassing *couples* subclasses to their superclass

Subclassing *enables* tight coupling in composed code

Don’t subclass.

— me, here, now

Approach 1:

Factor class hierarchies out

Encapsulate the concept that varies.

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

Factor out independent work

```
class Post {
    let title: String
    …
}

class Tweet: Post { … }

class XMLPost: Post {
    let XMLData: NSData
    let titlePath: XPath
    …
}

class RSS1Post: XMLPost { … }
class RSS2Post: XMLPost { … }
```

Factoring out independent work

`XMLPost`:

— tightly couples *data types* to *parsing strategies*
— introduces margin for error
— is inflexible to change

Factoring out independent work

```
struct XMLParser { … }

class RSS1Post: Post { … }
class RSS2Post: Post {
    init(data: NSData) {
        let parser = XMLParser(data)
        super.init(title: parser.evaluateXPath(…), …)
    }
}
```

Favour solutions which simplify the code base.

Approach 2:

Protocols, not superclasses

Protocols are interfaces

— just the *relevant* details: properties & methods

— Cocoa protocols:

1. behaviour: `NSCoding`, `NSCopying`, `UITableViewDelegate`
2. model: `NSFetchedResultsSectionInfo`, `NSFilePresenter`

Protocols are shared interfaces

```
class Post {
    let title: String
    …
}

class Tweet: Post { … }
class RSS1Post: Post { … }
class RSS2Post: Post {
    init(data: NSData) {
        let parser = XMLParser(data)
        super.init(title: parser.evaluateXPath(…), …)
    }
}
```

Using protocols to share interfaces

```
protocol PostType {
    var title: String { get }
    …
}

struct RSS2Post: PostType {
    let title: String
    …

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

Factor out independent interfaces

- `UITableViewDelegate` has >= 9 jobs (!)
- `...DataSource`/`...Delegate` are interdependent
- only used by & tightly coupled to `UITableView`
- forces implementing type to handle multiple concerns
- exact same problem as ill-factored classes

Delegate protocols suggest better factoring

- instead, *“encapsulate the concept that varies”*
  - factor out types for independent concerns (e.g. groups)
  - KVO-compliant selected/displayed subset properties (or signals) instead of `will…`/`did…`
  - 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 (many) minimal types

Post as a minimal type

```
struct Post {
    let title: String
    …
}

func ingestResponse(response: Response) -> Post? {
    switch response.contentType {
    case .RSS1:
        let parser = XMLParser(response.data)
        return Post(title: parser.evaluateXPath(…))
    default:
        return nil
    }
}
```

enums are fixed shared interfaces

Use enum for fixed sets of alternatives:

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

Minimal types are value types

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, Jamie Murai, Kelly Rix, Haleigh Sheehan, Justin Spahr-Summers, Patrick Thomson, and you ❤️