Codify the Pattern

How Swift lets us eschew convention

Codify the Pattern

Conventions get embeded in team culture and the app

Possibly documented:
* readme
* wiki
* Confluence

Swift makes it easy to codify conventions

Swift is opinionated

Objective-C Conventions vs Swift

Objective-C Convention	**Swift**
Initialization	`convenience` & `designated`
Mutable / Immutable types	`var` / `let`
nil, null, NSNull, -1	Optional
Properties -title, -setTitle:	var title { get {} set {} }

Maybe our code should be opionionated

— Me, just now

Rectangle Type

```
struct Rectangle {

    let identifier: String

    var x: Double
    var y: Double
    var width: Double
    var height: Double

    var red: Double
    var green: Double
    var blue: Double
    var alpha: Double
}
```

Atomic Types

Atomic Types

Separation of properties that are unlikely to change as one

Group properties that change together

Creating Atomic Types

```
struct Position {
    let x: Double
    let y: Double
}

struct Size {
    let width: Double
    let height: Double
}

struct Color {
    let red: Double
    let green: Double
    let blue: Double
    let alpha: Double
}
```

Creating Atomic Types

```
struct Rectangle {
    let identifier: String
    var position: Position
    var size: Size
    var color: Color
}
```

Changes to each property are atomic

Separation of concerns

Easier to reason about

These new types can be extended

Type Specialization

Type Specialization

```
struct Rectangle {
    let identifier: String
    var frame: Frame
    var color: Color
}
```

Identifier is a string so it can be used like a string

```
var identifier = rectangle.identifier
let start = identifier.index(after: identifier.startIndex)
let end = identifier.index(before: identifier.endIndex)
identifier.replaceSubrange(start..<end, with: "")
```

Type Specialization

```
struct Identifier {

    private let value: String

    init(_ value: String) {
        self.value = value
    }
}
```

Type Specialization

```
struct Identifier: Equatable {

    private let value: String

    init(_ value: String) {
        self.value = value
    }
}
```

Provide Equatable conformance to allow external use to determine equality rather than allow access to the value inside.

Type Specialization

```
struct Identifier: Equatable {

    private let value: String

    init(_ value: String = NSUUID().uuidString) {
        self.value = value
    }
}
```

Provide a standard way to generate new identifiers.

Type Specialization

```
struct Rectangle {

    let identifier = Identifier()

    var frame: Frame
    var color: Color
}
```

Constricting types

Constricting types

Developer is confused.

Developer attempts to create color.

```
rectangle.color = Color(red: 100, green: 50, blue: 0, alpha: 100)
```

A fundamental misundertanding has occured.

Validate with Fatal Error

```
struct Color {

    init(red: Double, green: Double, blue: Double, alpha: Double) {

        guard
            0 <= red, red <= 1,
            0 <= green, green <= 1,
            0 <= blue, blue <= 1,
            0 <= alpha, alpha <= 1
        else {
            fatalError()
        }

        ...
    }
}
```

Validate Using Minimum and Maximum Values

```
struct Color {

    init(red: Double, green: Double, blue: Double, alpha: Double) {

        switch red {
            case  ...0 : self.red = 0
            case 1...  : self.red = 1
            default    : self.red = red
        }

        switch green {
            case  ...0 : self.green = 0
            case 1...  : self.green = 1
            default    : self.green = green
        }

        ...
    }
}
```

Validate Using Specialized Type

```
struct Percentage {

    fileprivate let value: Double
    init(_ value: Double) {

        switch value {
            case  ...0 : self.value = 0
            case 1...  : self.value = 1
            default    : self.value = value
        }
    }
}
```

Validate Using Specialized Type

```
struct Color {
    let red: Percentage
    let green: Percentage
    let blue: Percentage
    let alpha: Percentage
}
```

Each color component now *must* be between zero and one.

Parameters define themselves as percentages.

Unfamiliar developers can look up the Percentage logic.

Validate Using Specialized Type

```
rectangle.color = Color(red: Percentage(2),
                        green: Percentage(0.5),
                        blue: Percentage(-1),
                        alpha: Percentage(100))
```

Components are validated before Color is created.

Expressible By Float/Integer Literal

```
extension Percentage: ExpressibleByFloatLiteral {

    init(floatLiteral value: Double) {
        self.init(value)
    }
}

extension Percentage: ExpressibleByIntegerLiteral {

    init(integerLiteral value: Int) {
        self.init(Double(value))
    }
}
```

Expressible By Float/Integer Literal

```
rectangle.color = Color(red: 2, green: 0.5, blue: -1, alpha: 100)

// Creates Color(red: 1, green: 0.5, blue: 0, alpha: 1)
```

Using integer and float *literal* values

Values from elsewhere still need to call Percentage initialiser

Makes for easier to read tests

Improving Size

```
struct Size {
    let width: Positive
    let height: Positive
}
```

This definition makes it clear what values are accepted

User of this type can inspect to verify their aassumptions of the logic of Positive

Improving Size

```
struct Positive {

    fileprivate let value: Double
    init(_ value: Double) {

        switch value {
            case  ...0 : self.value = 0
            default    : self.value = value
        }
    }
}
```

Preventing Invalid States

Adding Gradient

```
struct Gradient {
    let start: Color
    let end: Color
}
```

We've made sure Gradient is:

- Atomic
- A specialized type
- Can't represent an illegal state

Add Gradient to Rectangle

```
struct Rectangle {
    let identifier = Identifier()
    var position: Position
    var size: Size
    var fill: Color?
    var gradient: Gradient?
}
```

What happens if `fill` and `gradient` are both set?

What happens if they are both nil?

Preventing Invalid States

```
enum Style {
    case fill(color: Color)
    case linear(gradient: Gradient)
    case radial(gradient: Gradient)
}

struct Rectangle {
    let identifier = Identifier()
    var position: Position
    var size: Size
    var style: Style
}
```

Drawing a rectangle

```
switch rectangle.style {
case .fill(let color): // Fill with color
case .linear(let gradient): // Fill linear gradient
case .radial(let gradient): // Fill radial gradient
}
```

When we come to draw the rectangle, our logic for drawing is pre-defined.

Validating User Input

```
extension Percentage {

    init?(_ string: String) {

        guard let double = Double(string) else {
            return nil
        }

        self.init(double)
    }
}
```

Optional initialiser prevents an invalid Percentage being created

But what was the issue?

```swift
extension Percentage {

    enum Error: Swift.Error {
        case invalidCharacter, unknown
    }

    init(_ string: String) throws {

        let validSet = CharacterSet(charactersIn: "-.0123456789")
        let set = CharacterSet(charactersIn: string)
        guard set.isSubset(of: validSet) else {
            throw Error.invalidCharacter
        }

        guard let double = Double(string) else {
            throw Error.unknown
        }

        self.init(double)
    }
}
```

The following gives a valid Percentage

```
do {
    let percentage = try Percentage("4")
} catch {
    print(error)
}
```

The following throws an invalid character error

```
do {
    let percentage = try Percentage("A")
} catch {
    print(error) // "invalidCharacter"
}
```

The following throws an unknown error

```
do {
    let percentage = try Percentage("0.0.0")
} catch {
    print(error) // "unknown"
}
```

Restricting APIs

Incorrect Initializers

Types can be extended and custom initializers added:

```
extension Percentage {

    init(badValue: Double) {
        self.value = badValue
    }
}
```

Use modules to prevent misuse

If an initializer is declared in a different module from a struct, it must use self.init(...) or self = ... before returning or accessing self

— SE-0189, Restrict Cross-module Struct Initializers

This prevents custom initializers in another module creating invalid instances.

Final Thoughts

Final Thoughts

Raises questions about **edge cases** upfront

Most business logic happens in the model

Actually simpler to change incorrect assumptions

Can split out JSON -> model types conversion to another module

Daniel Tull

@danielctull