A Swift Journey

Luke Stringer

Hi, I'm Luke 👋

@lukestringer90

I'm the Head of Mobile 3Squared

- Software for Rail and Construction industries
- Based in Sheffield
- Build Web apps
  - C# .NET
- Build native iOS and Android apps
  - Swift, Objective-C & Java
- I lead team of 7 mobile developers

Our "Swift" Journey

for 2 reasons...

1. We moved from Objective-C to Swift

2. We did it Quickly

Our 2016 Goals

1. Train the mobile team in Swift
2. Move from Obj-C to Swift for new projects
3. Develop reusable Swift code
4. Modernise & standardise our tools and practises

1. Training

Looking for Training

We needed training materials that could be used for:

- New / Inexperiences developers - placement students
- Experienced non-iOS developers - Android developers
- Experienced iOS developers

Styles of Learning

- Follow guided tutorials
- Build a simple project in Swift
  - Request data from GitHub API
  - Parse & persist
  - Display on screen
- Build a wrapper and other reusable code

Just Do It!

Learn whilst Building Something Real

Learning & Discussion as a Team was Crucial

2. New Projects in Swift

First Project

100% Swift from the Start!

First Project

- Chose a small / medium complexity project
- Evaluated Swift's readiness for production
- Discussed code design and style as a team
- Identified gaps in our knowledge & our tools

Results

- Project delivered on time
- No major problems - Swift is production ready
- Beginnings of an agreed code style
- Built helpers for using Core Data & Storyboards

The Next Projects

Following Code Style, using Helpers

Next Projets

- 2 new projects 100% Swift from start
- More ambitions in scope and longer in duration
- More of the team involved
- Followed code style
- Used helper code from first project

Trialling Reusable Code

- At this stage reusable code was unversioned
- Just dropped straight into an Xcode project
- Provided freedom to adapt and change
- However this led to code divergence
- At a suitable point team decided to standardise the approaches

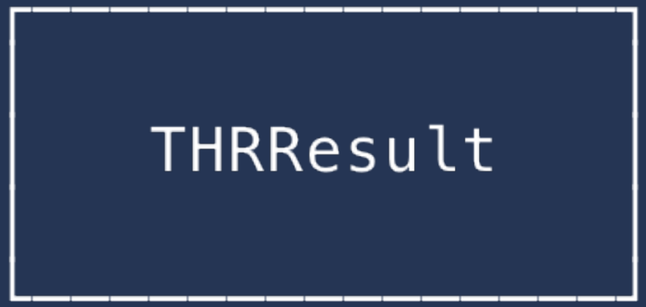
Results

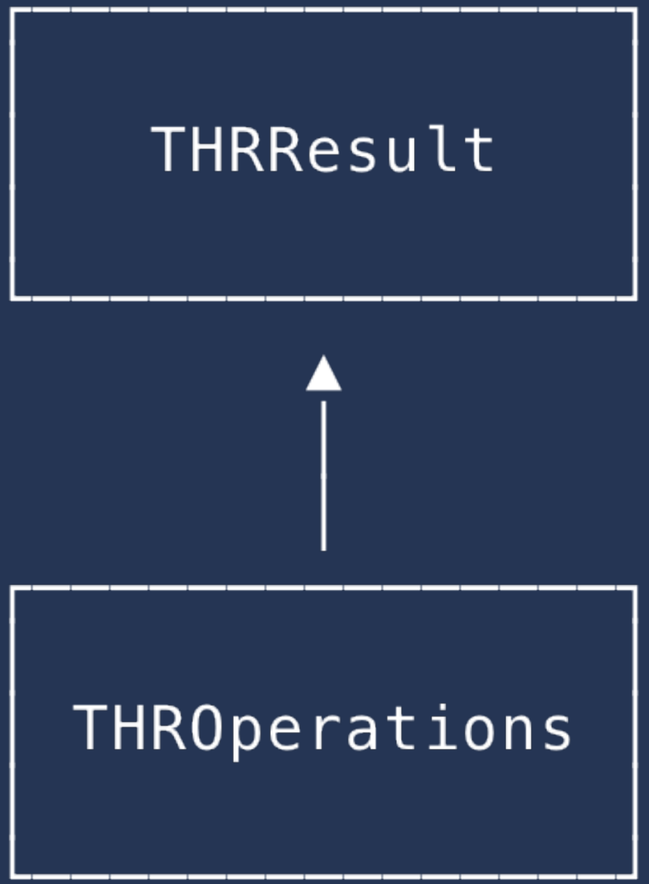
- Both projects delivered on time
- Code style adapted
- Filled more gaps - Networking, Threading, Error Handling
- Helper code adapted and now versioned using cocoapods

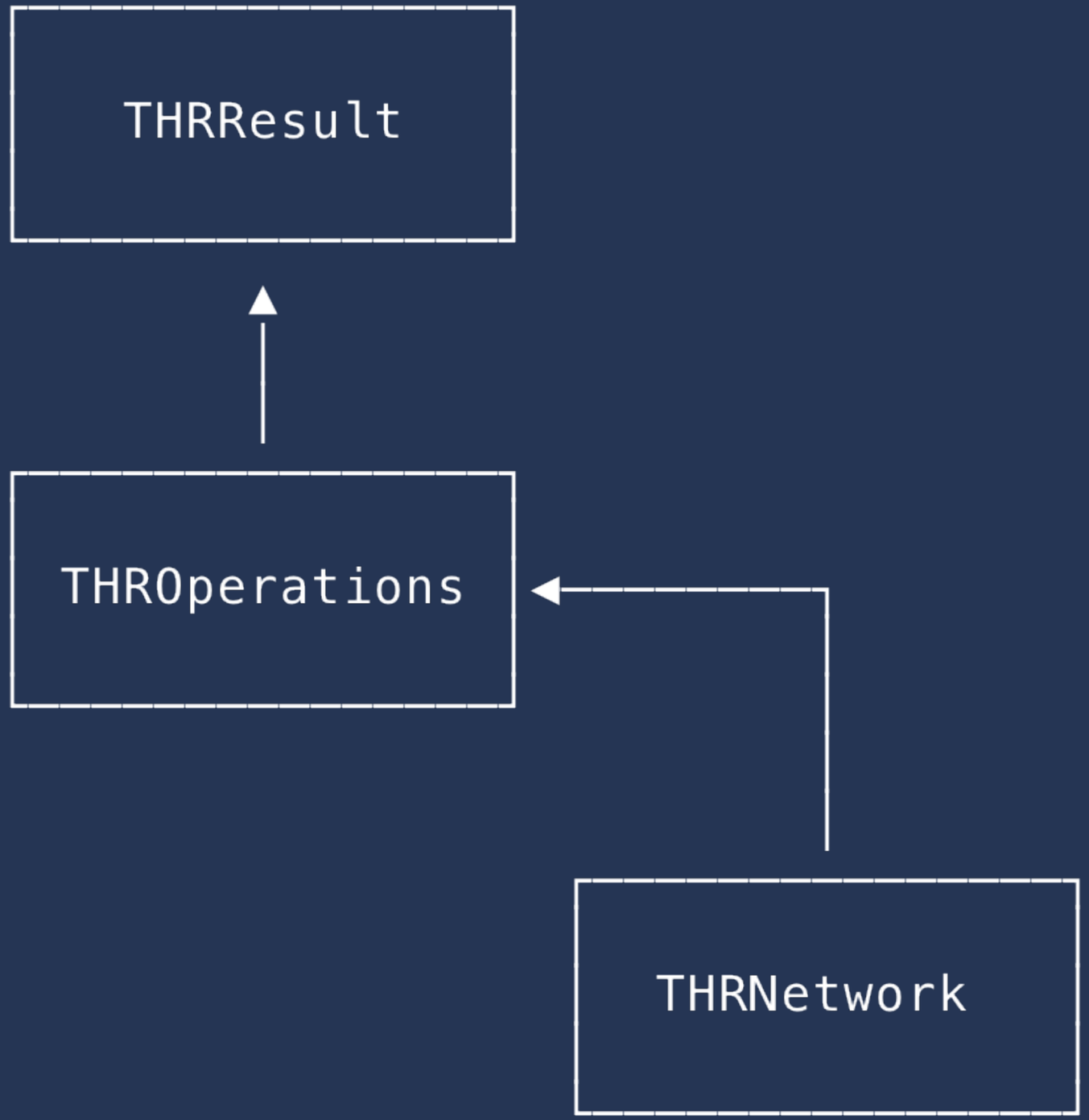
3. Reusable Code

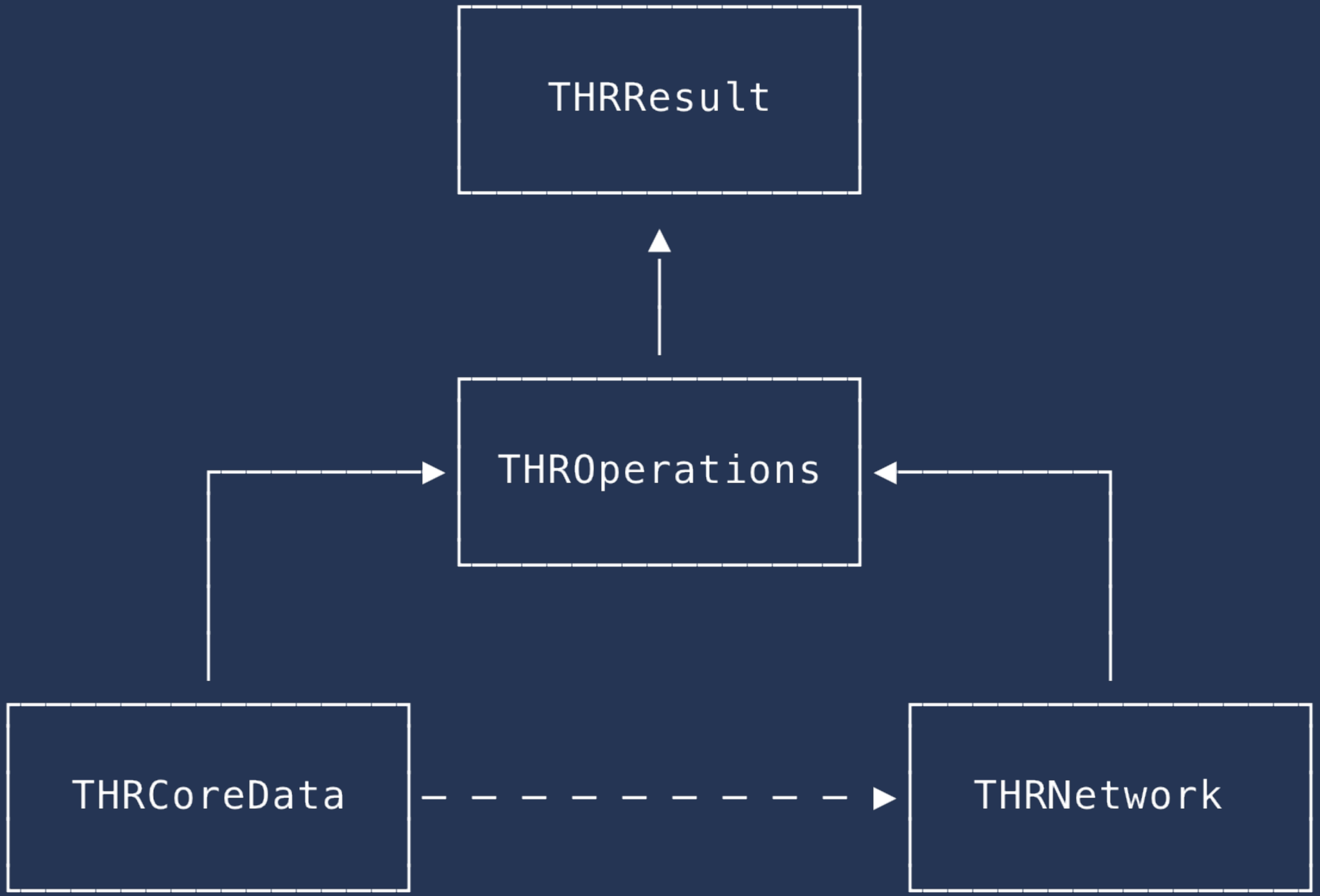
The "THR" Micro Frameworks

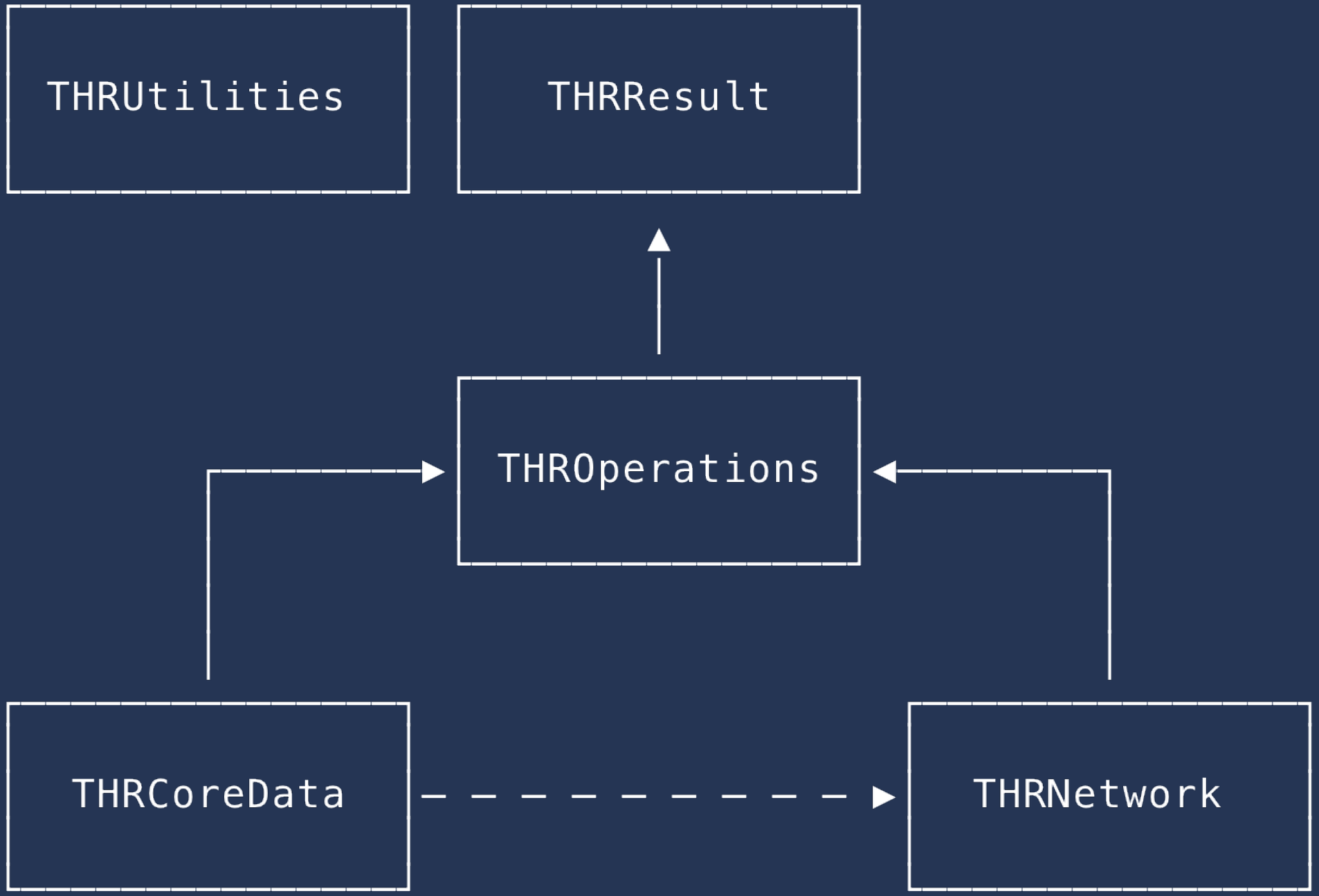
(Inspired by other works)

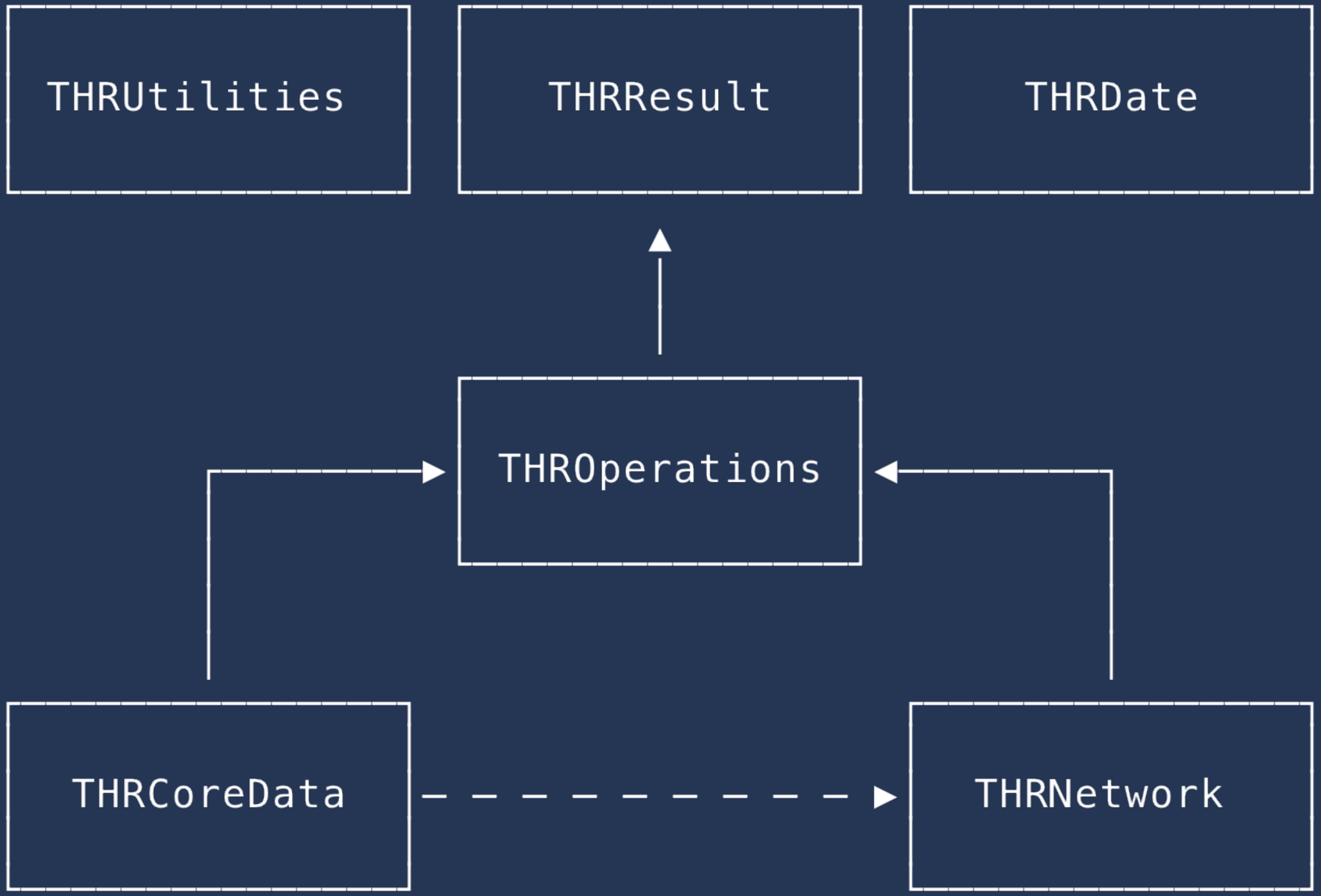

THRResult


THRResult
THROperations


THRResult
THROperations
THRNetwork


THRResult
THROperations
THRCoreData
THRNetwork


THRUtilities
THRResult
THROperations
THRCoreData
THRNetwork


THRUtilities
THRResult
THRDate
THROperations
THRCoreData
THRNetwork

`THRResult`

- Models success and failure
- Extends the familiar `Result` type to work with do-catch error handling
- Powerful, flexible and expressive
- Credit to
  - https://github.com/antitypical/Result
  - Olivier Halligon "Asynchronous error handling"

```
enum Result<T> {
    case success(T)
    case failure(Error)
}
```

```
enum Result<T> {
    case success(T)
    case failure(Error)
}

// Using a switch:
switch result {
case .success(let value):
    // process success value
case .failure(let error):
    // handle failure error
}
```

```swift
enum Result<T> {
    case success(T)
    case failure(Error)

    init(_ throwingExpr: () throws -> T) {
        do {
            let value = try throwingExpr()
            self = .success(value)
        } catch {
            self = .failure(error)
        }
    }

    func resolve() throws -> T {
        switch self {
        case .success(let value): return value
        case .failure(let error): throw error
        }
    }
}
```

```
let successResult = Result { return value }
let failureResult = Result { throw ImportError.invalidData }
```

```swift
let successResult = Result { return value }
let failureResult = Result { throw ImportError.invalidData }

// Using do-catch:
do {
    let value = try result.resolve()
    // Process value
}
catch {
    // Handle error
}
```

```swift
let successResult = Result { return value }
let failureResult = Result { throw ImportError.invalidData }

// Using if let
if let value = try? result.resolve() {
    // Process value
}

// or
let value = try! result.resolve()
```

`THROperations`

- Preferred method for threading
- Helpful subclasses of `Operation`
- Expressive syntax for combining operations
- Uses `THRResult` to report success and failure
- Credit to
  - https://github.com/ProcedureKit/ProcedureKit

```
protocol ProducesResult: class {
    associatedtype Output
    var output: Result<Output> { get set }
}
```

```
extension ProducesResult where Self: Operation {
    func addResultBlock(block: @escaping (Result<Output>) -> Void) {
        if let existing = completionBlock {
            completionBlock = {
                existing()
                block(self.output)
            }
        }
        else {
            completionBlock = {
                block(self.output)
            }
        }
    }
}
```

Example

Let's build an operation to extract a top level domain (TLD) from a URL string

```
typealias TopLevelDomain = String
typealias URLString = String
```

```
class ExtractTopLevelDomainOperation: Operation, ProducesResult {
    // ProducesResult
    var output: Result<TopLevelDomain> = Result {
        throw ResultError.noResult
    }
    // TopLevelDomainOperation
    let urlString: URLString
    init(urlString: URLString) {
        self.urlString = urlString
    }
    override func main() {
        super.main()
        extractTopLevelDomain(from: urlString)
    }
    // ...
}
```

```
// ...continued
func extractTopLevelDomain(from urlString: URLString) {
    guard urlString.characters.count > 0 else {
        output = Result { throw TopLevelDomainError.invalidURL }
        return
    }

    let parts = urlString.components(separatedBy: ".")

    guard parts.count > 1, let topLevelDomain = parts.last else {
        output = Result { throw TopLevelDomainError.none }
        return
    }

    output = Result { return topLevelDomain }
}
```

```
let operation = ExtractTopLevelDomainOperation(urlString: "apple.com")

operation.addResultBlock { result in
    switch result {
    case .success(let value):
        // process TLD
    case .failure(let error):

        switch error {
        case TopLevelDomainOperation.none:
            // handle no TLD
        case TopLevelDomainOperation.invalidURL:
            // handle invalid URL
        }
    }
}

operation.enqueue()
```

```
let operation = ExtractTopLevelDomainOperation(urlString: "apple.com")

operation.addResultBlock { result in
    do {
        let topLevelDomain = try result.resolve()
        // process TLD
    }
    catch {
        switch error {
        case TopLevelDomainOperation.none:
            // handle no TLD
        case TopLevelDomainOperation.invalidURL:
            // handle invalid URL
        }
    }
}

operation.enqueue()
```

```
let operation = ExtractTopLevelDomainOperation(urlString: "apple.com")

operation.addResultBlock { result in
    if let topLevelDomain = try? result.resolve() {
        // process TLD
    }
}

operation.enqueue()
```

Combining Operations

- Often the output from one operation is the input to another
- We want to pass the `Result` onwards
- Combining many simple operations can produce complex logic

```
protocol ProducesResult: class {
    associatedtype Output
    var output: Result<Output> { get set }
}

protocol ConsumesResult: class {
    associatedtype Input
    var input: Result<Input> { get set }
}
```

```
class ExtractTopLevelDomainOperation: BaseOperation,
                                      ProducesResult,
                                      ConsumesResult {
    // ProducesResult
    var output: Result<TopLevelDomain> = Result {
        throw ResultError.noResult
    }
    // ConsumesResult
    var input: Result<URLString> = Result {
        throw ResultError.noResult
    }

    // ...
}
```

```
// ...continued
override func main() {
    super.main()
    do {
        let urlString = try input.resolve()
        extractTopLevelDomain(from: urlString)
    }
    catch {
        output = Result { throw error }
    }
}
```

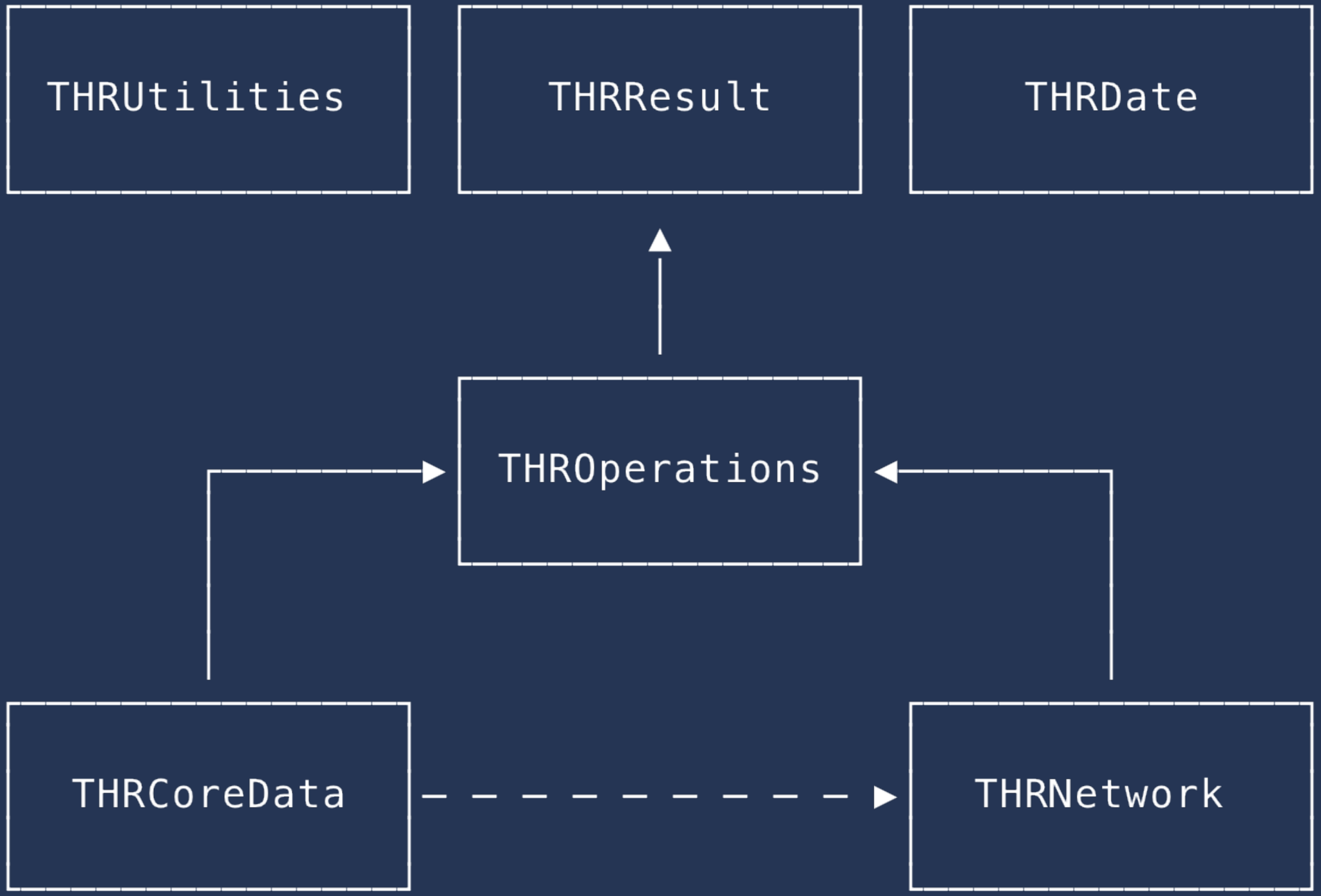
```
// Output: URLString
let getURLString = RequestURLStringOperation()

// Input: URLString, Output: TopLevelDomain
let extractTLD = ExtractTopLevelDomainOperation()

// Input: TopLevelDomain, Output: Bool
let validate = ValidateTopLevelDomainOperation()

validate.addResultBlock { result in
    if let isValid = try? result.resolve(), isValid == true {
        // TLD is valid
    }
}

getURLString
    .passesResult(to: extractTLD)
    .passesResult(to: validate)
    .enqueue()
```


THRUtilities
THRResult
THRDate
THROperations
THRCoreData
THRNetwork

4. Modernise & Standardise

Modernise & Standardise

- Handle errors and unhappy paths (`THRResult`)
- Perform work asynchronously (`THROperations`)
- Build complex logic from smaller units (`THROperations`)
- Request data over a network, then parse into core data (`THRNetwork` & `THRCoreData`)
- Use storyboard segues with type safety (`THRUtilities`)
- Parse dates to and from a web API (`THRDate`)

Did We Meet Our Goals?

Our 2016 Goals

1. Train the mobile team in Swift
2. Move from Obj-C to Swift for new projects
3. Develop reusable Swift code
4. Modernise & standardise our tools and practises

Our 2016 Goals

1. Train the mobile team in Swift ✅
2. Move from Obj-C to Swift for new projects
3. Develop reusable Swift code
4. Modernise & standardise our tools and practises

Our 2016 Goals

1. Train the mobile team in Swift ✅
2. Move from Obj-C to Swift for new projects ✅
3. Develop reusable Swift code
4. Modernise & standardise our tools and practises

Our 2016 Goals

1. Train the mobile team in Swift ✅
2. Move from Obj-C to Swift for new projects ✅
3. Develop reusable Swift code ✅
4. Modernise & standardise our tools and practises

Our 2016 Goals

1. Train the mobile team in Swift ✅
2. Move from Obj-C to Swift for new projects ✅
3. Develop reusable Swift code ✅
4. Modernise & standardise our tools and practises ✅

Recommendations

1. Learn as a Team

2. Gain Support from Business Management

3. Make Space for Experimentation

4. Expect to make Mistakes

Our Plan for The Future

How do we stop writing Objective-C in existing apps?

Every line of Objective-C written now is Technical Debt

Swift alongside Objective-C

- Understand the limitations of Obj-C/Swift interoperability
  - No `structs`
  - No associated values with enums
  - No protocol extensions
- Identify new features that can be written in Swift without *too much* difficulty

Swift alongside Objective-C

- Replace existing standalone Obj-C classes, methods etc with Swift implementation
  - `NSPredicate` generation
  - Simple / isolated `UIViewControllers`
- Replace existing Obj-C structures with `THR` frameworks
  - e.g. rewrite data synchronisation subsystem

A Swift Journey

Luke Stringer

Any Questions? 

References

- http://alisoftware.github.io/swift/async/error/2016/02/06/async-errors/
- https://github.com/antitypical/Result
- https://github.com/ProcedureKit/ProcedureKit