These are not the slides you’re looking for…

App Frameworks

#WWDC15

Advanced NSOperations

Exploring the WWDC app

Session 226

Philippe Hausler Foundation Engineer
Dave DeLong Frameworks Evangelist

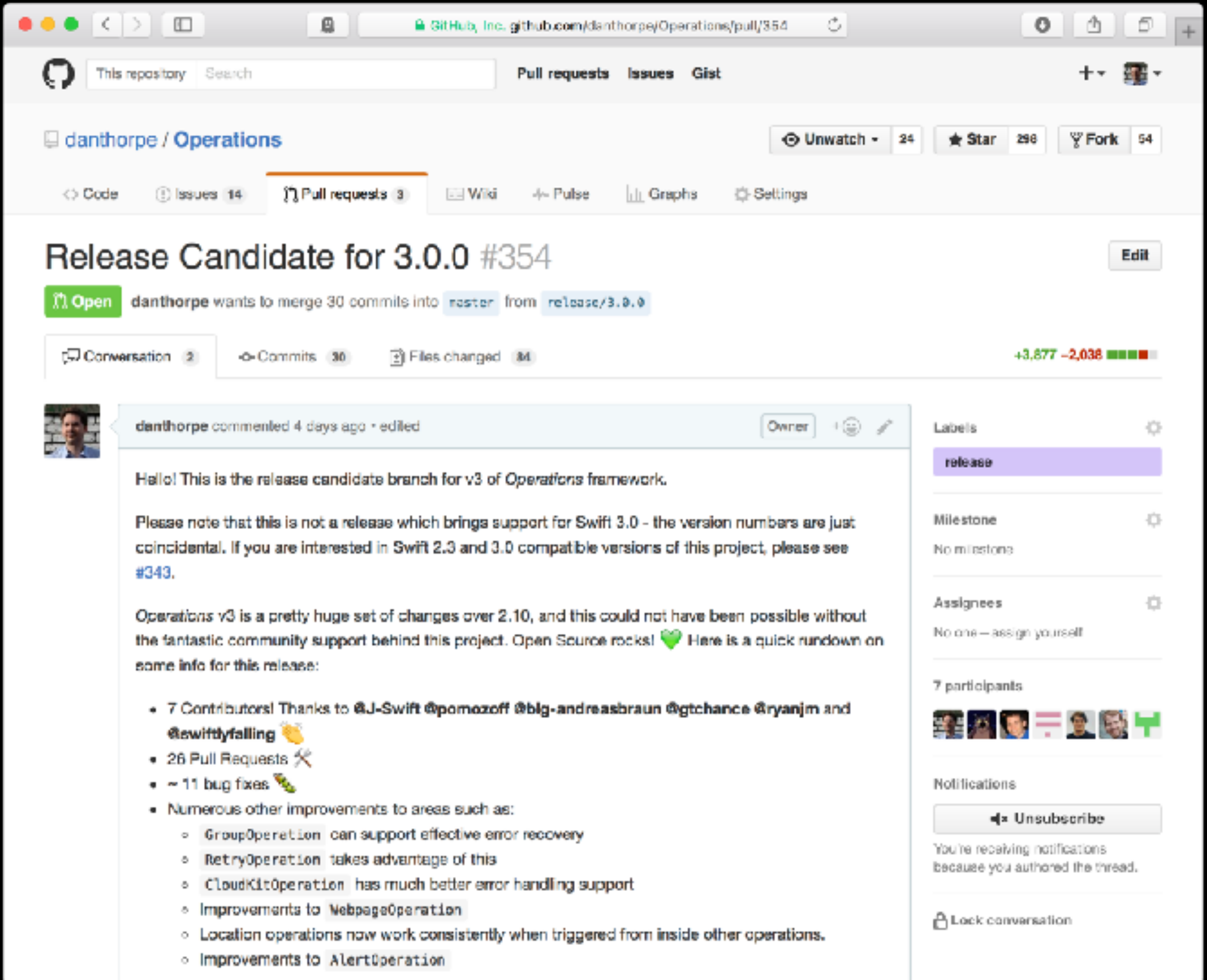
Advanced, Advanced Operations

@danthorpe

danthorpe/Operations

https://github.com/danthorpe/

Operations



Release Candidate for 3.0.0 #354

Open danthorpe wants to merge 30 commits into `master` from `release/3.0.0`

danthorpe commented 4 days ago • edited

Hello! This is the release candidate branch for v3 of *Operations* framework.

Please note that this is not a release which brings support for Swift 3.0 - the version numbers are just coincidental. If you are interested in Swift 2.3 and 3.0 compatible versions of this project, please see #343.

Operations v3 is a pretty huge set of changes over 2.10, and this could not have been possible without the fantastic community support behind this project. Open Source rocks! 💚 Here is a quick rundown on some info for this release:

- 7 Contributors! Thanks to **@J-Swift @pomozoff @big-andreasbraun @gtchance @ryanjm** and **@swiftlyfalling** 👏
- 26 Pull Requests 🛠
- ~ 11 bug fixes 🐛
- Numerous other improvements to areas such as:
  - `GroupOperation` can support effective error recovery
  - `RetryOperation` takes advantage of this
  - `CloudKitOperation` has much better error handling support
  - Improvements to `WebpageOperation`
  - Location operations now work consistently when triggered from inside other operations.
  - Improvements to `AlertOperation`

Agenda

Very Quick Recap

Advanced Operations

Beyond the Basics

Practical use case @ Sky

Recap

NSOperationQueue

High-level abstraction of **dispatch_queue_t**

Supports cancellation

Variable width

NSOperation

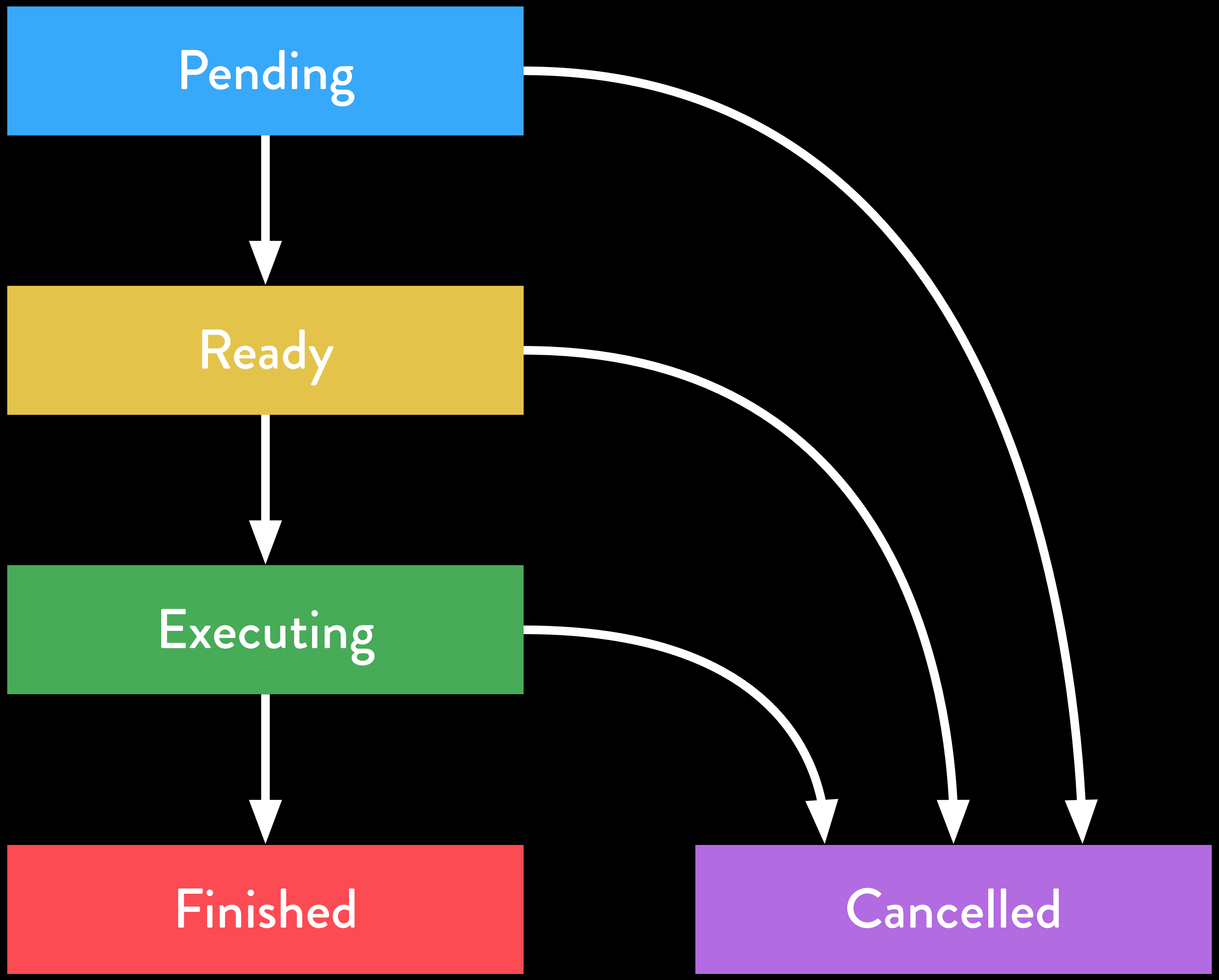
High-level abstraction of **dispatch_block_t**

Supports long running tasks

Supports object oriented design

Well defined lifecycle

NSOperation Lifecycle



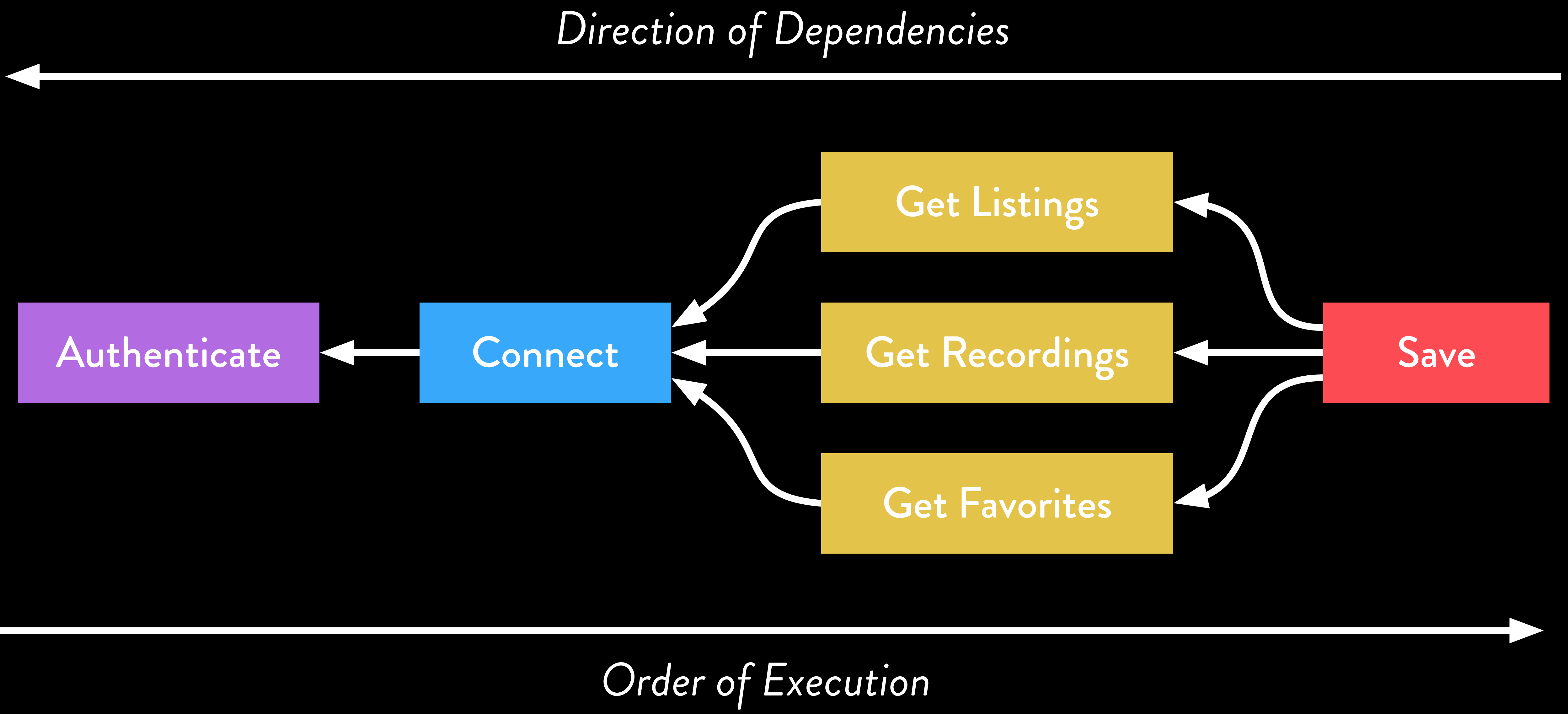
Abstraction

Operations abstract and isolate business logic

Encourages a decoupled and component-based architecture

Encourages code re-use

Dependencies



Dependencies

Defines a strict ordering between operations

An operation will become ready, when all of its dependencies have finished

Works across operation queues

Advanced Operations

Finished vs Failed

NSOperation instances always finish

No definition of success or failure

Operation supports the concept of finishing with errors

An operation which fails has still finished

Conditions

Conditions express the requirements needed to execute an operation

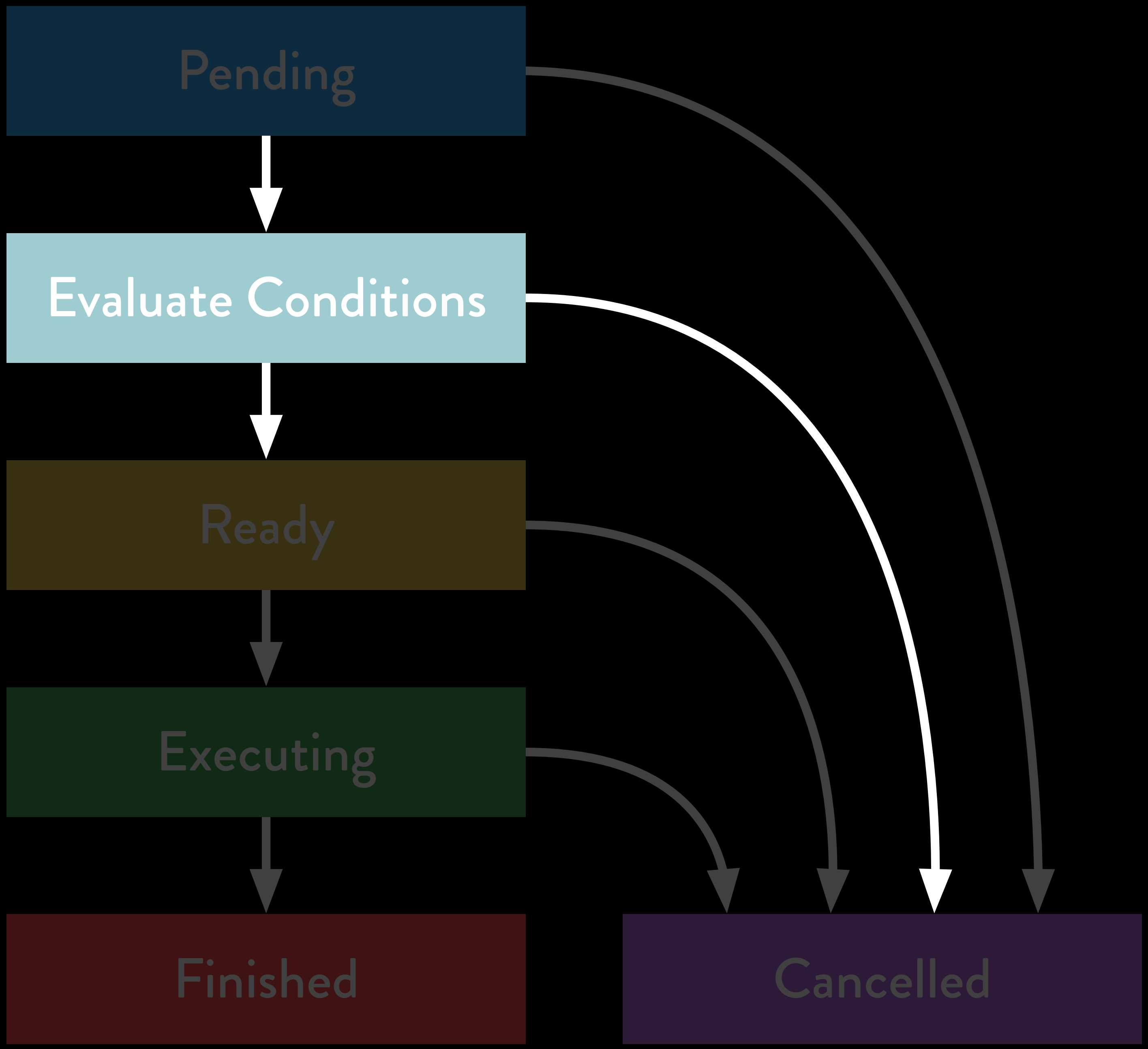
Conditions can “gate” functionality based on outside factors or business logic

Conditions are attached to an Operation

Conditions are evaluated *asynchronously* to either succeed or fail with an error

Conditions are Operations

How *did* conditions work?



Evolution

Evaluating Conditions

Must override isReady

Begin evaluating conditions when super is ready

Sets ready state asynchronously from with its getter

Very subtle bugs in NSOperation

Do not override isReady

Conditions

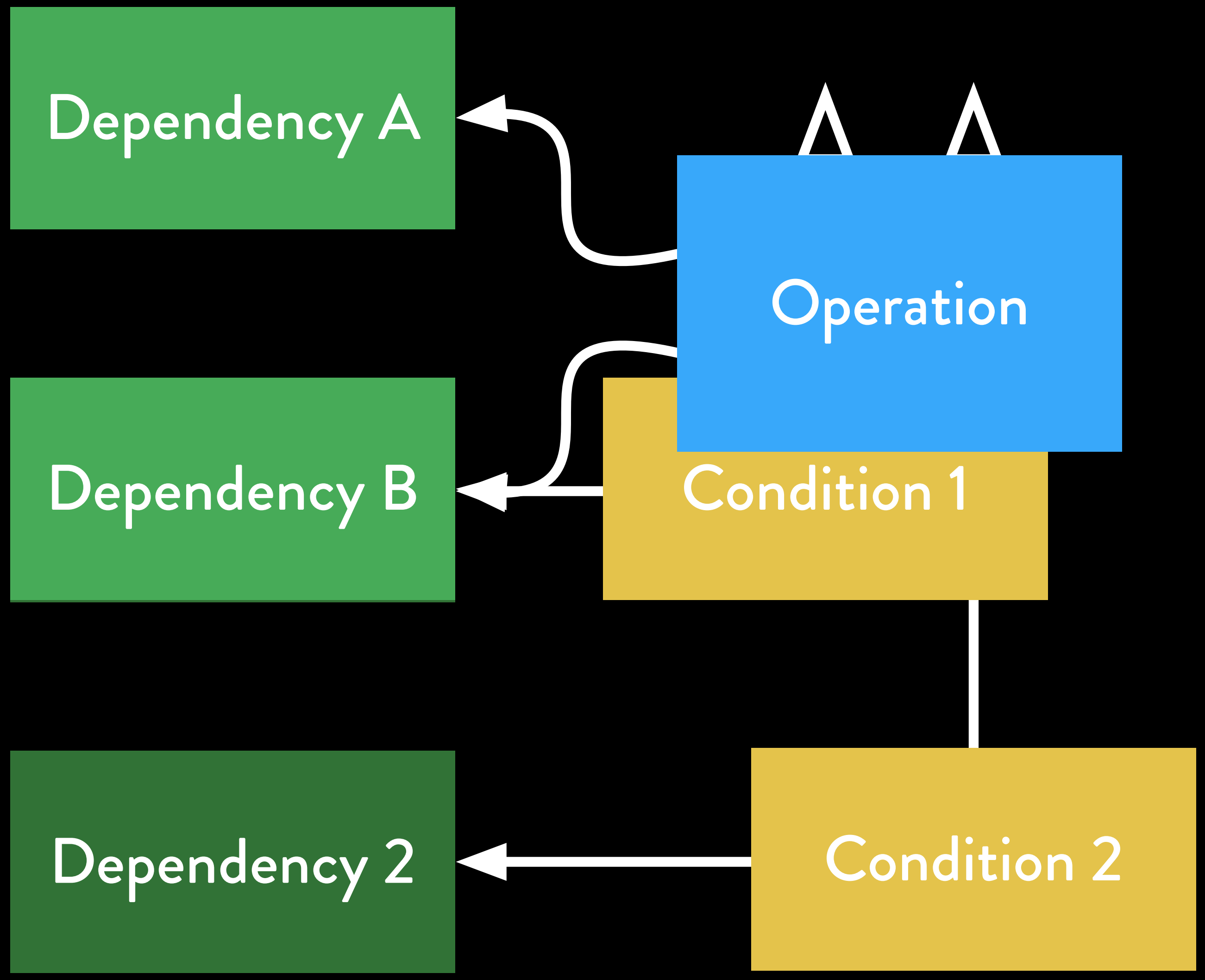
Conditions are Operations

Add like a regular dependency

Avoids overriding isReady

Target operation waits for all its conditions to evaluate before becoming ready

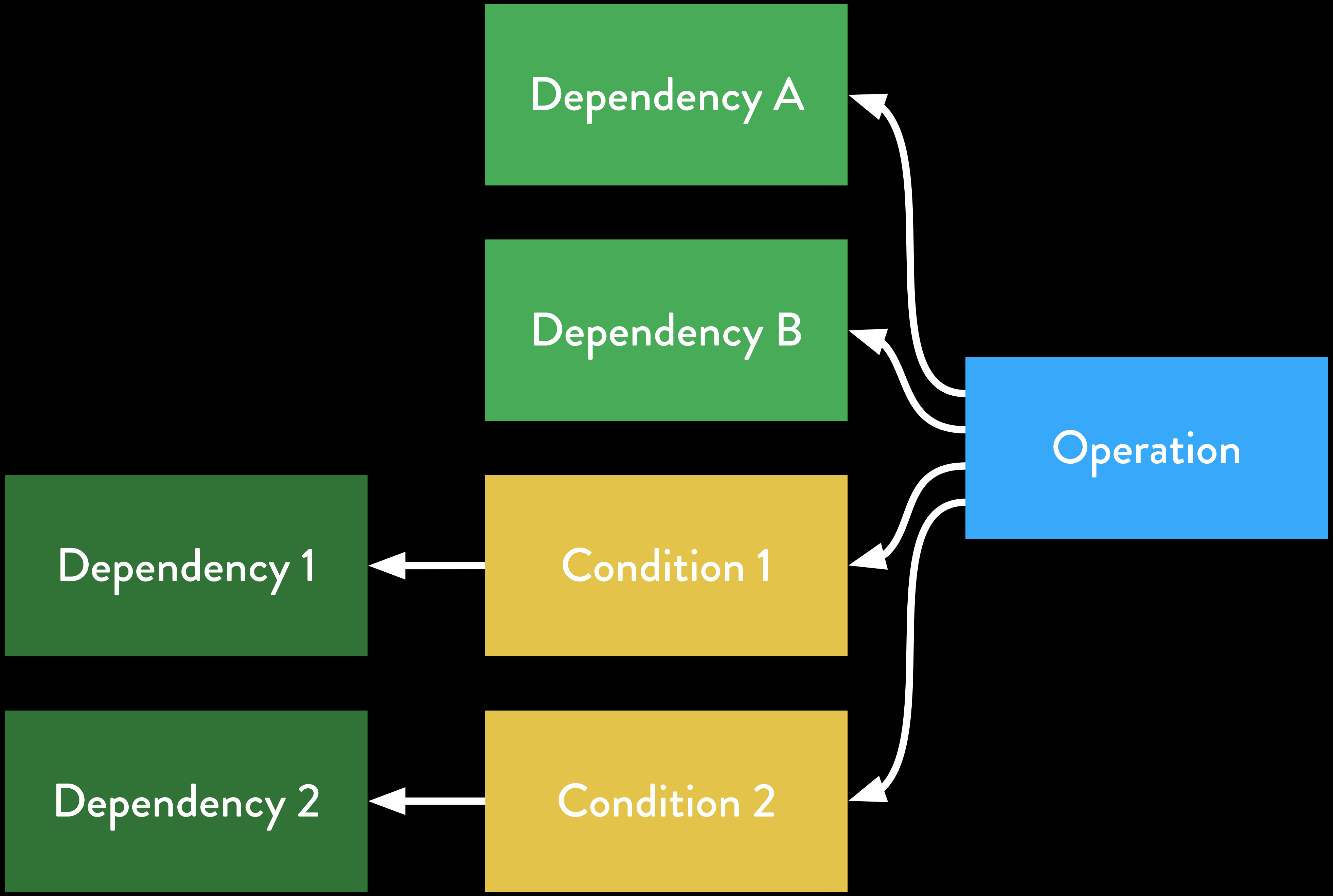
Conditions & Dependencies



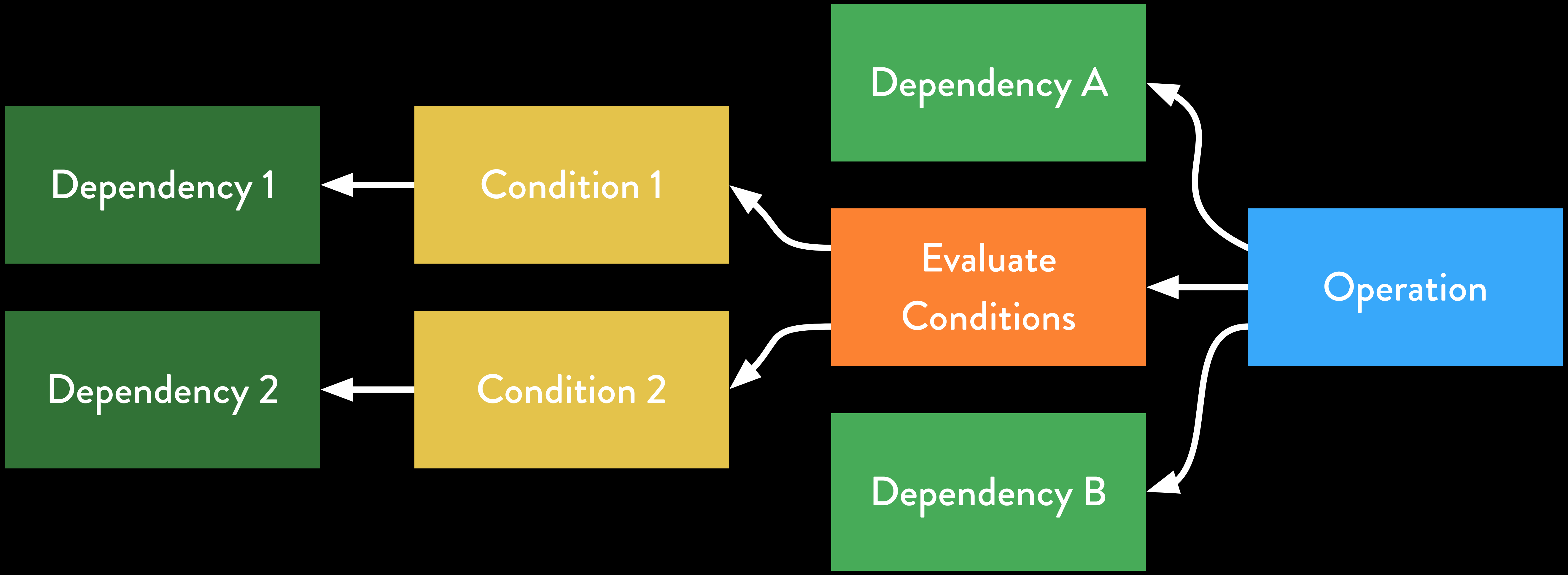
Conditions & Dependencies

```swift
let dependencyA = DependencyA()
let dependencyB = DependencyB()

let operation = MyOperation()
operation.addDependencies([dependencyA, dependencyB])

operation.addCondition(Condition1())
operation.addCondition(Condition2())

queue.addOperations(operation, dependencyA, dependencyB)
```

Scheduling

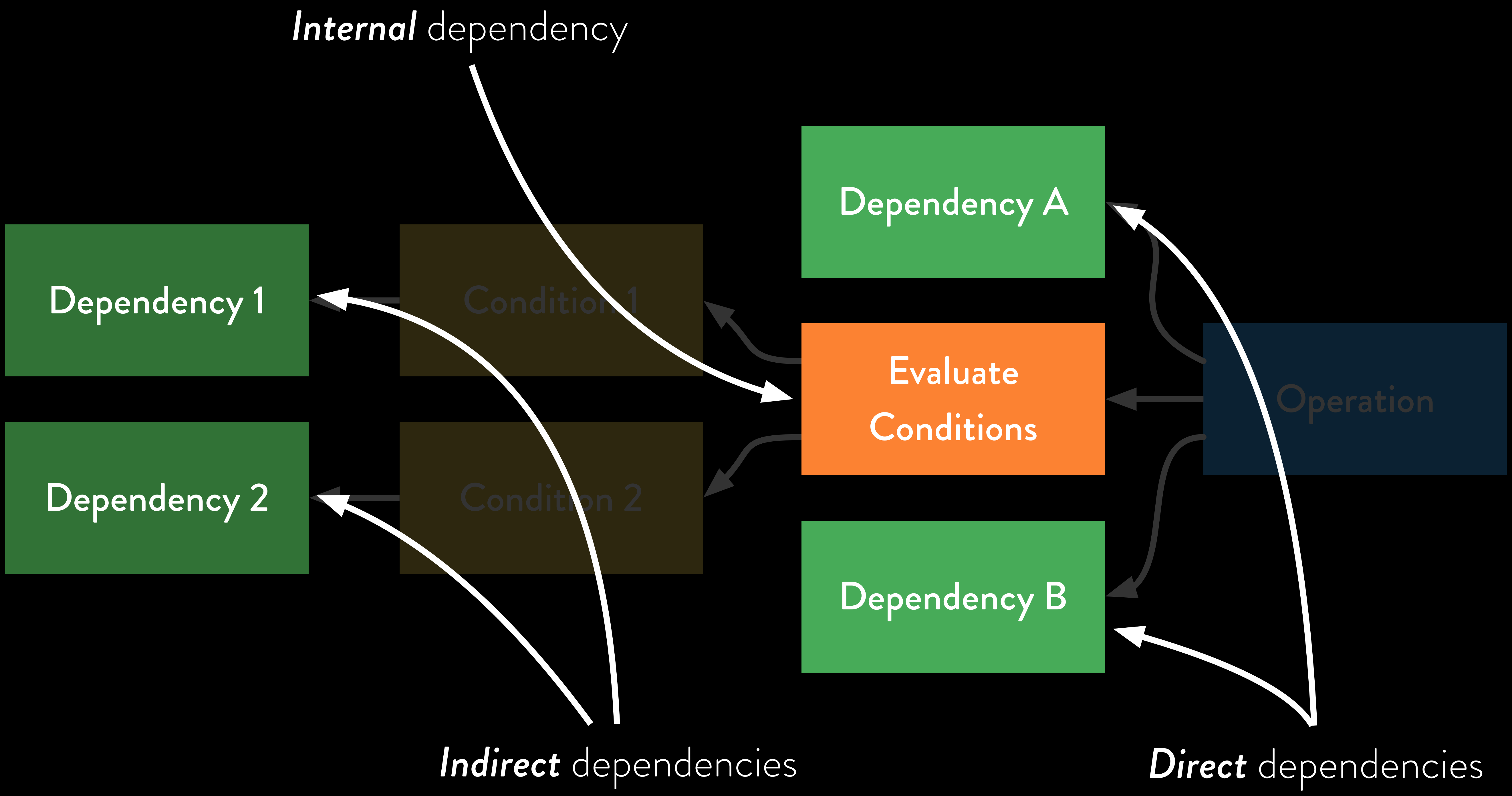

Dependency A
Dependency B
Operation
Dependency 1
Condition 1
Dependency 2
Condition 2

Scheduling



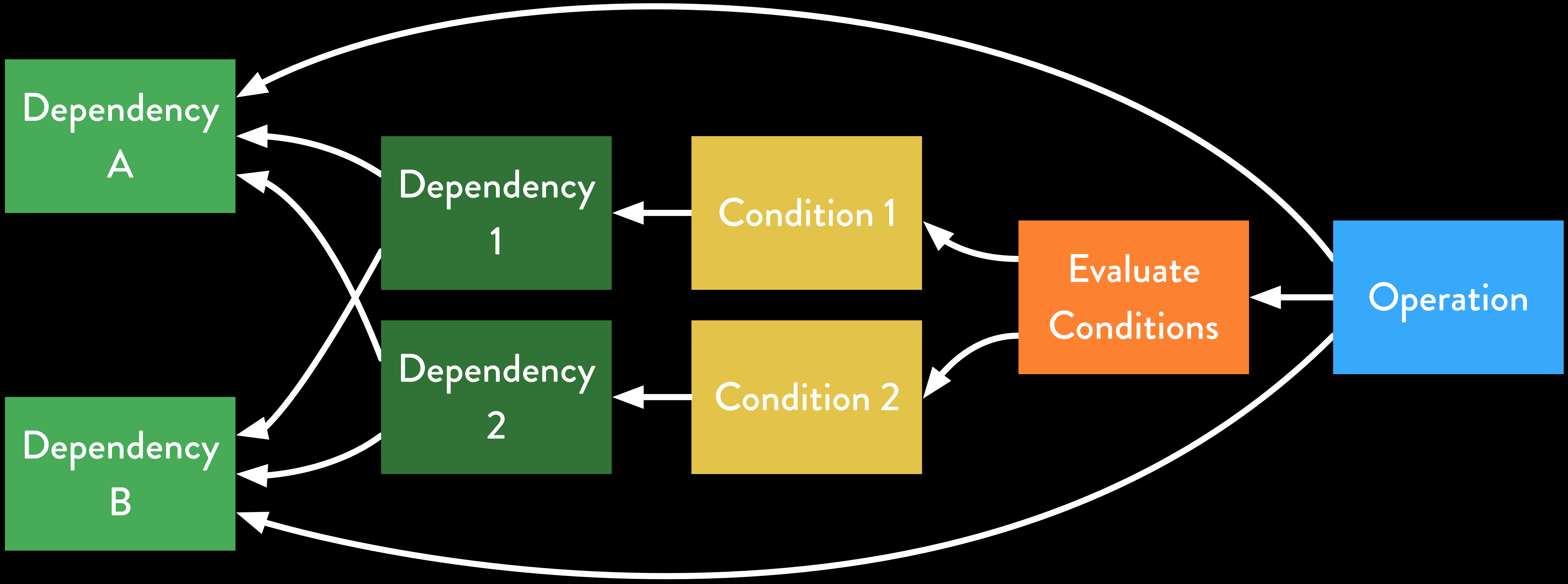
Dependency A
Dependency 1
Condition 1
Evaluate Conditions
Operation
Dependency 2
Condition 2
Dependency B

Scheduling



Internal dependency
Dependency A
Dependency 1
Condition 1
Evaluate Conditions
Operation
Dependency 2
Condition 2
Dependency B
Indirect dependencies
Direct dependencies

Scheduling


Dependency A
Dependency B
Dependency 1
Dependency 2
Condition 1
Condition 2
Evaluate Conditions
Operation

Scheduling

Operation has ***direct*** dependencies, just like NSOperation

Additionally, if it has conditions, it creates and depends on an evaluator of the conditions.

All ***indirect*** dependencies depend on all direct dependencies

Mutual Exclusivity

Prevent operations with the same category from running simultaneously

Alerts or modal UI presentations

Restrict access to resources

Operation depends on the previous exclusive operation

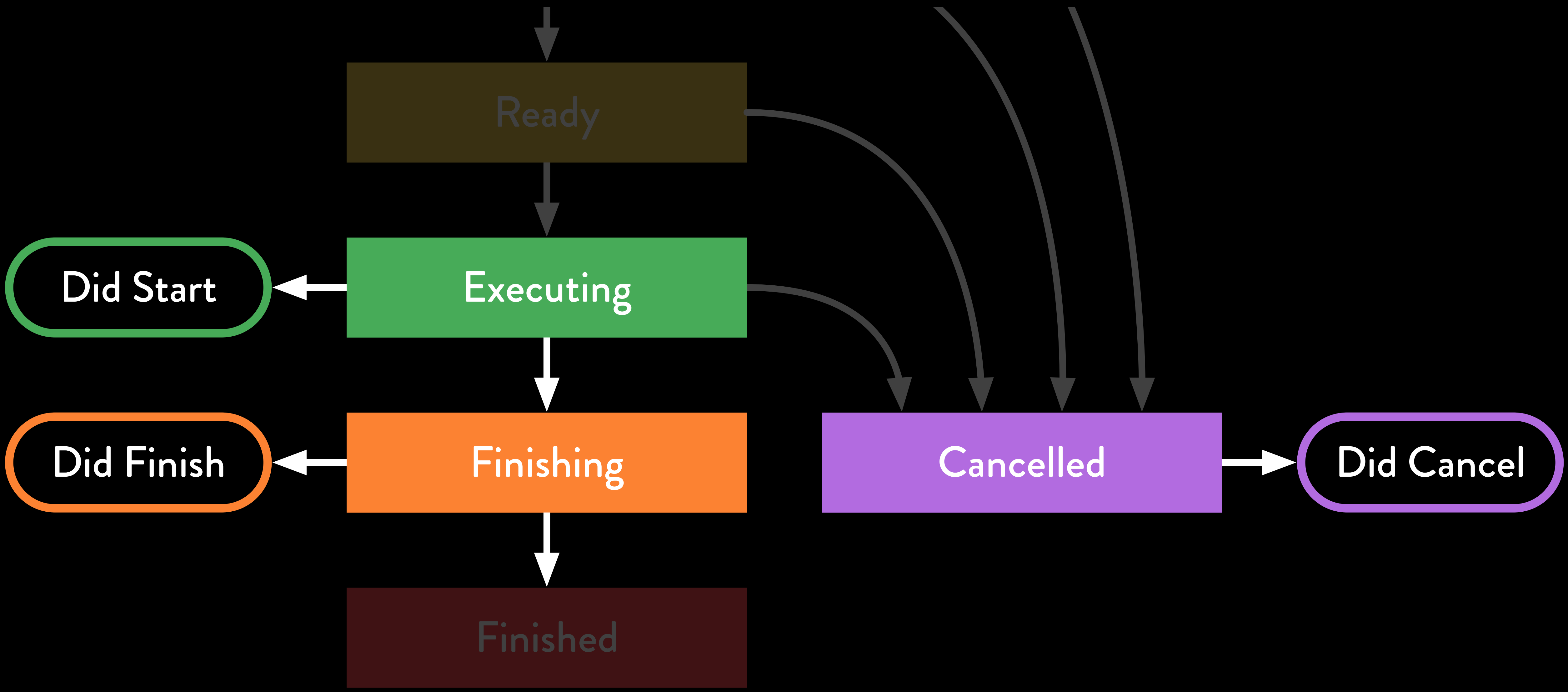
Observers

Observers are attached to an Operation

When the operation transitions between states it invokes the relevant observers

Observer pattern decouples action/reaction logic

How *did* observers work?



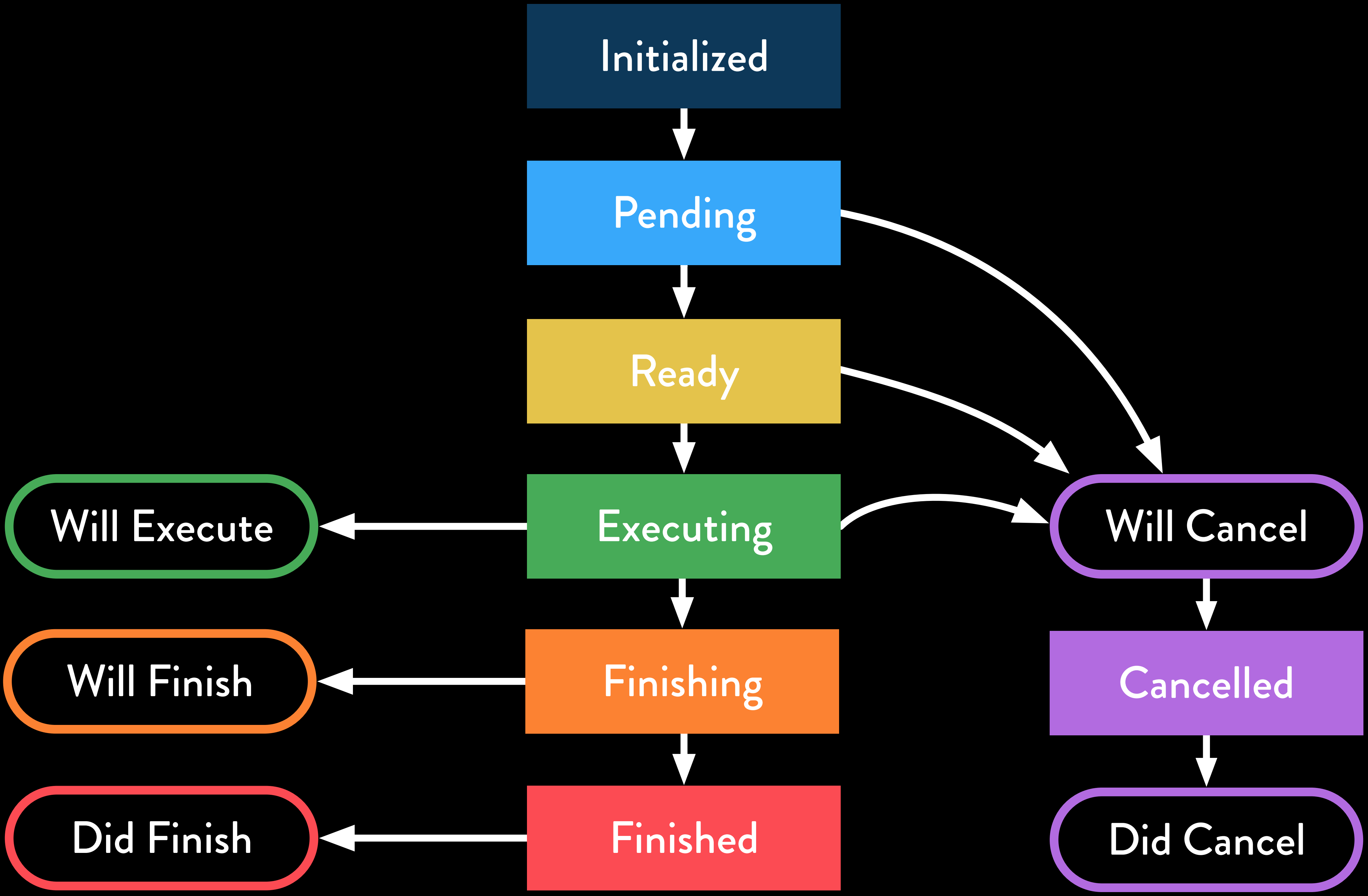
Triggering Observers

Single protocol with multiple methods

The observer must be added to the operation before the operation begins executing

Works very well for system observers such as Network Activity Indicator

Observers



Initialized
Pending
Ready
Executing
Will Execute
Finishing
Will Finish
Finished
Did Finish
Will Cancel
Cancelled
Did Cancel

Observers

Base observer protocol

Can be safely added to Operation instances at any point in their lifecycle

Each protocol supports one Operation state transition

Full complement of concrete types which take blocks

Beyond the Basics

Advanced Usage: Profiler

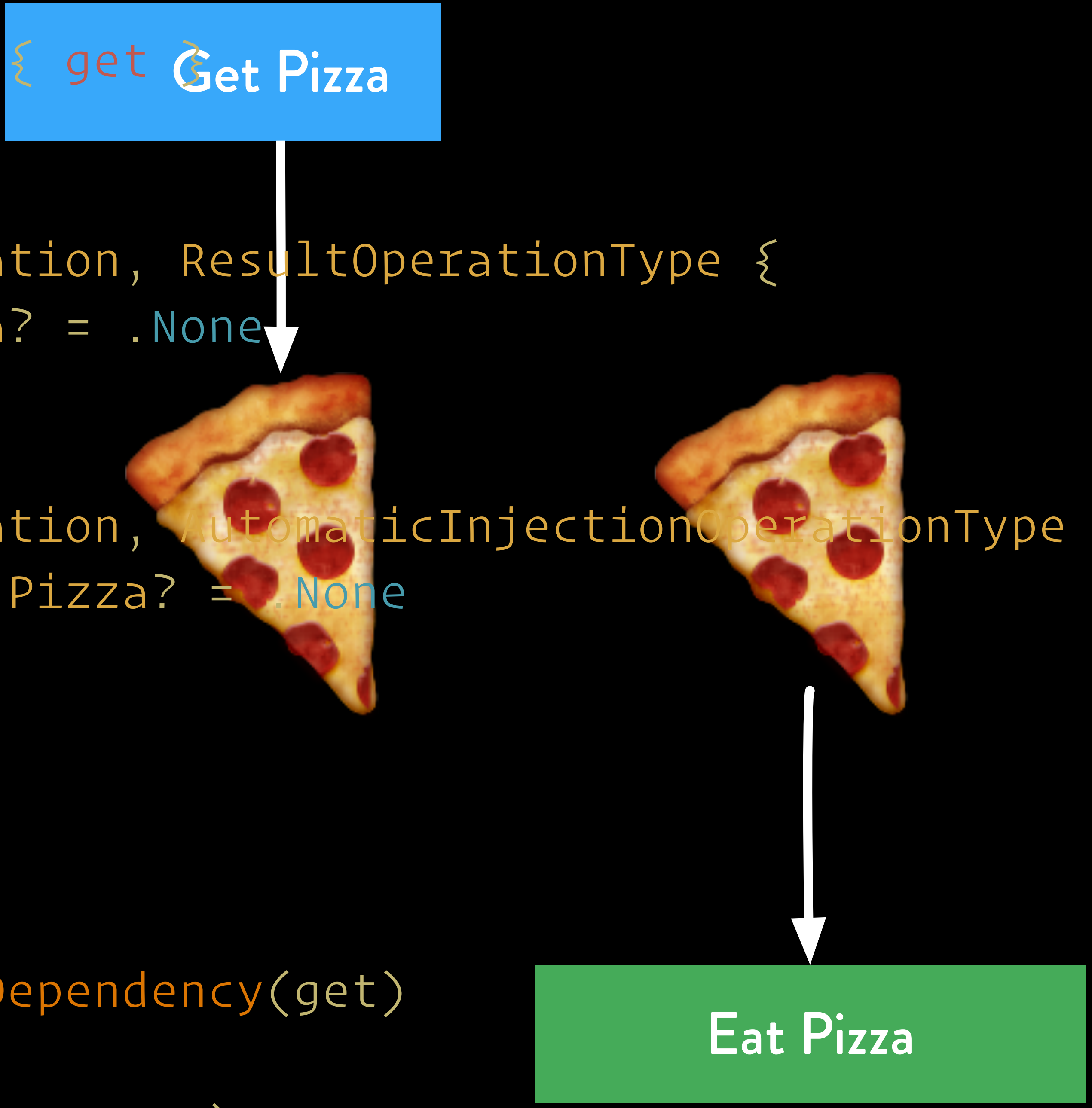
An observer to profile operations with support for custom reporters.

It can time transition between states e.g. how long did the operation run for? How long did it spend waiting?

Will automatically attach itself to any produced or child operations to profile entire graphs of operations

Result Injection

```
protocol Pizza {
    var name: String { get }
}

class GetPizza: Operation, ResultOperationType {
    var result: Pizza? = .None
}

class EatPizza: Operation, AutomaticInjectionOperationType {
    var requirement: Pizza? = .None
}

let get = GetPizza()
let eat = EatPizza()

eat.injectResultFromDependency(get)

queue.addOperations(get, eat)
```



Result Injection

Conform to InjectionOperationType

```
operation.injectResultFromDependency(dependencyC) {
    operation, dependency, errors in
    // This block is executed before the operation
    // starts. It can be used to access properties of
    // the dependency.
}
```

Adds observers to the dependency

Automatically cancels operation if dependency is cancelled

Sets up dependency relationship

Result Injection

```swift
protocol Pizza {
    var name: String { get }
}

class GetPizza: Operation, ResultOperationType {
    var result: Pizza? = .None
}

class EatPizza: Operation, AutomaticInjectionOperationType {
    var requirement: Pizza? = .None
}

let get = GetPizza()
let eat = EatPizza()

eat.injectResultFromDependency(get)

queue.addOperations(get, eat)
```

Result Injection

Types which ***produce a result*** should conform to ResultOperationType

Types which ***consume a result*** as their ***requirement*** must conform to AutomaticInjectionOperationType

Built in generic operations for map, filter and reduce.

Group Operation

Operation subclass which manages its own queue

Can further abstract logic into smaller operation types

Supports adding child operations after it begins

Excellent pattern for greedy operations

Adding operations to Groups

Subclass GroupOperation

Override *willFinishOperation(operation: NSOperation)*

Optionally override *recoveredFromErrors(errors: [ErrorType], inOperation: NSOperation)*

Call *addOperation(newOperation)*

Repeated Operation

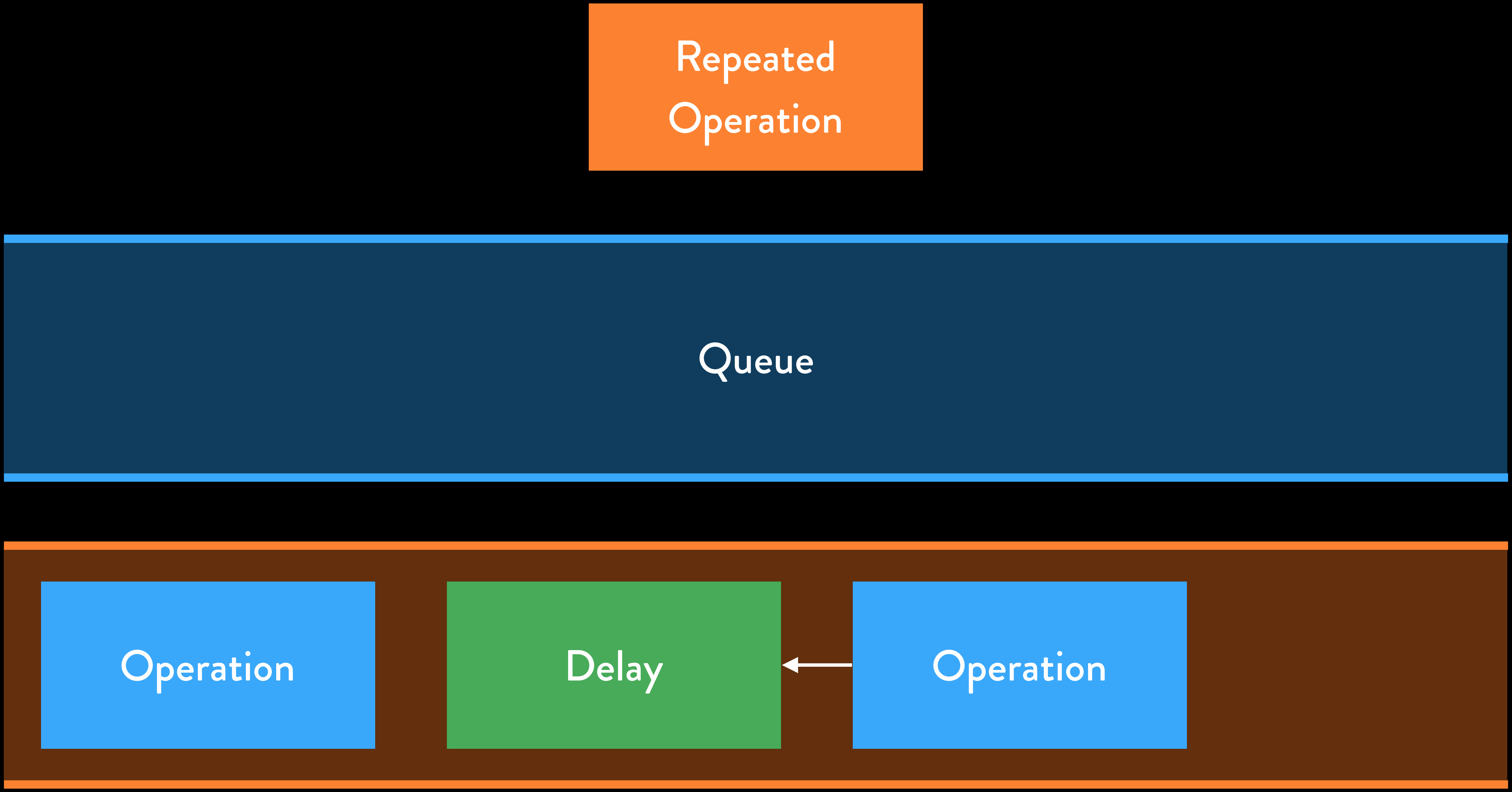
Generic GroupOperation subclass to support invoking the same operation type over and over

Initialised with a Generator

End repeating by returning nil from generator, or initialising with max count value

Supports arbitrary delays between instances of the operation

Repeated Operation



Wait Strategy

RepeatedOperation can add a delay between executing each generated operation instance.

Can initialise with a WaitStrategy to automatically generate fixed, random, incrementing, exponential or fibonacci delays.

Create functionality which exponentially backs-off

Retry Operation

RepeatedOperation subclass

Will repeat the operation in the event of an error

Supports a custom error handler to modify the instance of the retried operation

CloudKitOperation is a RetryOperation subclass with built in error handling for many CloudKit errors

More…

Capabilities

A ***Capability*** defines access to a ***resource***

The resource will likely have its own ***status*** type

The access may have a ***requirement*** at granular levels

Capabilities

CapabilityType is a generic protocol

Checks if capability is available

Get the current status

Request authorisation

Default implementation for Location, Calendar, Photos, Passbook, HealthKit, CloudKit

Using Capabilities

```
func locationServicesEnabled(enabled: Bool, withAuthorization status:
CLAuthorizationStatus) {
    // etc switch over (enabled, status) to update UI
}

func determineAuthorizationStatus() {
    let status = GetAuthorizationStatus(Capability.Location(), completion:
locationServicesEnabled)
    queue.addOperation(status)
}

func requestPermission() {
    let authorize = Authorize(Capability.Location(), completion:
locationServicesEnabled)
    queue.addOperation(authorize)
}
```

AuthorizedFor<Capability>

AuthorizedFor is a Condition which does all the heavy lifting for you

Given a capability, it will add Authorize(capabilty) as a dependency

When it evaluates, it will check the status and whether the requirements were met

If not, the condition will fail with a CapabilityError

Composing Conditions

NegatedCondition will negate the result of the composed condition

```
let condition = AuthorizedFor(Capability.Location())
op.addCondition(NegatedCondition(condition))
```

SilentCondition removes the dependencies from its composed condition

```
op.addCondition(SilentCondition(NegatedCondition(condition)))
```

Custom Capability

Log into custom network backend

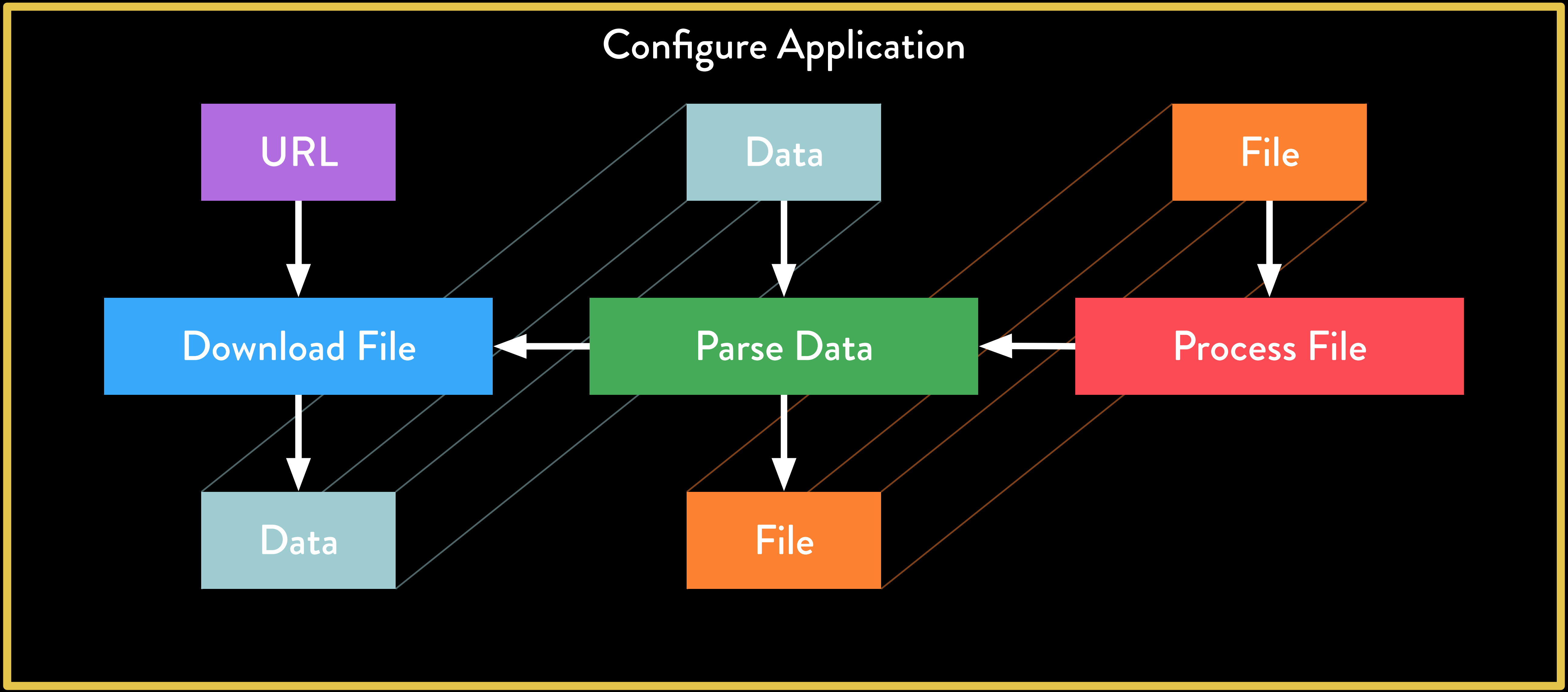
OS features, maybe Capability.TouchID()

Feature Flags

Set-top-box entitlements

Application Features / Gatekeeper

Practical use case @ Sky



```swift
class ConfigureApplication: GroupOperation, AutomaticInjectionOperationType {

    struct File: ConfigurationFile { … }

    var requirement: NSURL?
    let download: DownloadFile

    init(URL: NSURL? = .None) {
        requirement = URL

        download = DownloadFile(URL: URL)

        let parse = ParseData<File>()
        parse.injectResultFromDependency(download)

        let process = ProcessFile<File>()
        process.injectResultFromDependency(parse)

        super.init(operations: [download, parse, process])
        name = "Configure Application"
        addCondition(MutuallyExclusive<ConfigureApplication>())
    }

    override func execute() {
        guard let URL = requirement else { finish(Error.URLNotProvided); return }
        download.requirement = URL
        super.execute()
    }
}
```

```swift
class DownloadFile: Operation, AutomaticInjectionOperationType, ResultOperationType {

    var requirement: NSURL?
    var result: (NSData, NSURLResponse)? = .None

    var session: NSURLSession = NSURLSession.sharedSession()

    init(URL: NSURL? = .None) {
        requirement = URL
        super.init()
        name = "Download File"
        addObserver(NetworkObserver())
    }

    override func execute() {
        guard let URL = requirement else { finish(Error.URLNotProvided); return }

        session.dataTaskWithURL(URL) { [unowned self] data, response, error in

            guard let data = data, response = response else {
                let e = error.map { Error.DownloadError($0) } ?? Error.EmptyResponse
                self.finish(e)
                return
            }

            self.result = (data, response)

            self.finish()
        }
    }
}
```

```swift
protocol ConfigurationFile {
    init(data: NSData) throws
    func process(_: ErrorType? -> Void)
}

class ParseData<File: ConfigurationFile>: Operation, AutomaticInjectionOperationType,
ResultOperationType {

    var requirement: (NSData, NSURLResponse)?
    var result: File?

    override func execute() {

        guard let data = requirement?.0 else {
            finish(Error.DataNotProvided); return
        }

        guard let file = try? File(data: data) else {
            finish(Error.NotWellFormedData); return
        }

        result = file

        finish()
    }
}
```

```
class ProcessFile<File: ConfigurationFile>: Operation,
AutomaticInjectionOperationType {

    var requirement: File?

    override func execute() {
        guard let file = requirement else {
            finish(Error.MissingFile); return
        }
        file.process(finish)
    }
}
```

```
class ConfigureApplication: GroupOperation, AutomaticInjectionOperationType {

    struct File: ConfigurationFile { ··· }

    var requirement: NSURL?
    let download: DownloadFile

    init(URL: NSURL? = .None) {
        requirement = URL

        download = DownloadFile(URL: URL)

        let parse = ParseData<File>()
        parse.injectResultFromDependency(download)
        parse.addCondition(NoFailedDependenciesCondition())

        let process = ProcessFile<File>()
        process.injectResultFromDependency(parse)
        process.addCondition(NoFailedDependenciesCondition())

        super.init(operations: [download, parse, process])
        name = "Configure Application"
        addCondition(MutuallyExclusive<ConfigureApplication>())
    }

    override func execute() {
        guard let URL = requirement else { finish(Error.URLNotProvided); return }
        download.requirement = URL
        super.execute()
    }
}
```

```swift
class ConfigureApplication: GroupOperation, AutomaticInjectionOperationType {

    struct File: ConfigurationFile { ⋯ }

    var requirement: NSURL?
    let download: RetryOperation<DownloadFile>

    init(URL: NSURL? = .None) {
        requirement = URL

        download = RetryOperation(strategy: .Exponential((period: 30, maximum: 300)), AnyGenerator { DownloadFile(URL: URL) })

        let parse = ParseData<File>()
        parse.injectResultFromDependency(download) { operation, dependency, errors in
            guard let result = dependency.result else {
                operation.cancelWithError(Error.DataNotProvided); return
            }
            operation.requirement = result
        }

        let process = ProcessFile<File>()
        process.injectResultFromDependency(parse)
        process.addCondition(NoFailedDependenciesCondition())

        super.init(operations: [download, parse, process])
        name = "Configure Application"

        addCondition(MutuallyExclusive<ConfigureApplication>())
    }

    override func execute() {
        guard let URL = requirement else { finish(Error.URLNotProvided); return }

        download.requirement = URL

        super.execute()
    }
}
```

Operations

Encapsulate work into a re-usable component

Combine isolated components together to reduce complexity

Complex scheduling is easily expressed via dependencies, conditions, exclusivity and groups

Handle errors via groups and observers at the appropriate level of abstraction

Thanks for listening!

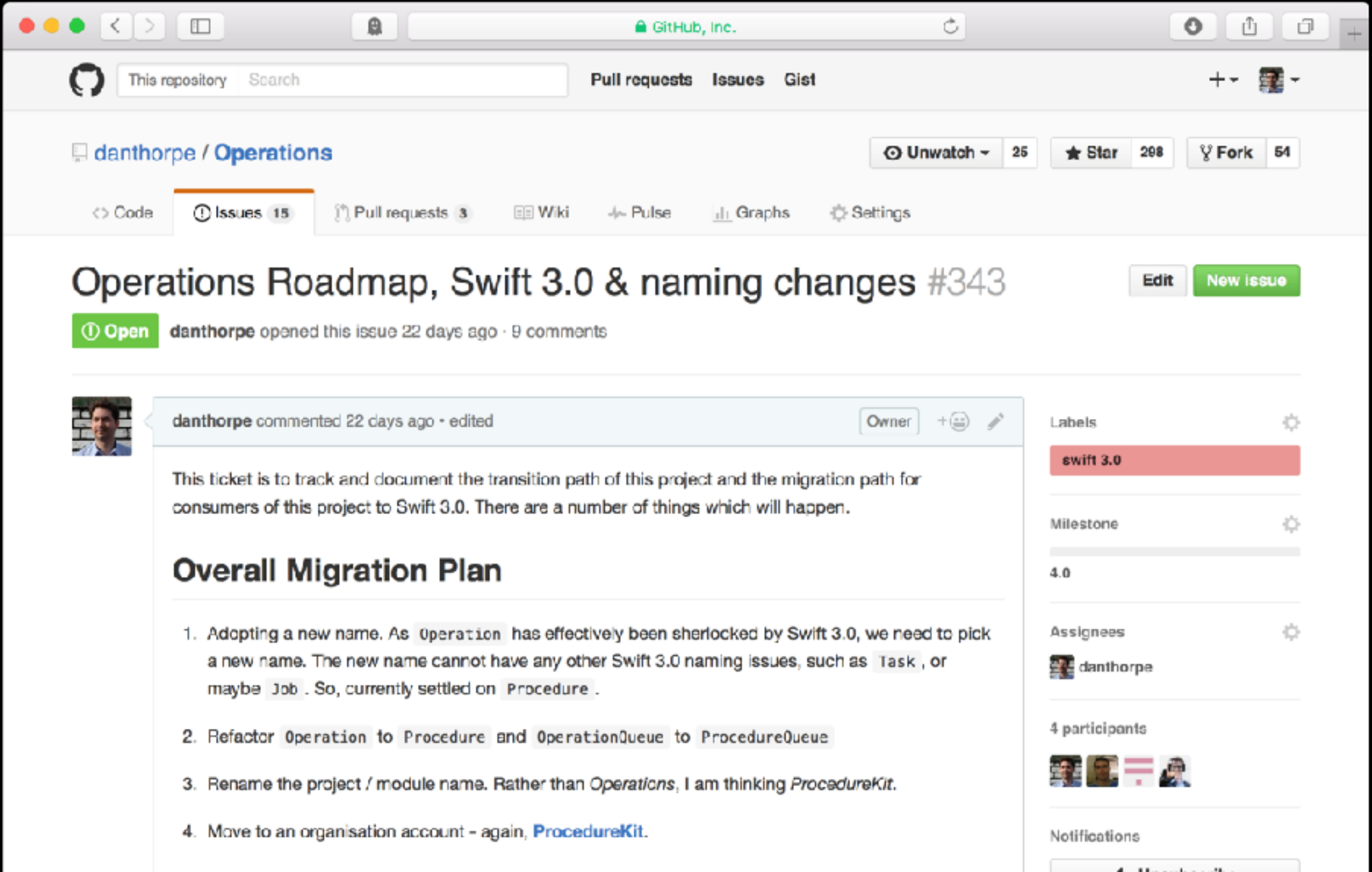
@danthorpe

danthorpe/Operations

danthorpe@me.com

With 💚 thanks to all the contributors and consumers of Operations so far! 😀

Swift 3.0 Migration


GitHub, Inc.
This repository Search
Pull requests Issues Gist
danthorpe / Operations
Unwatch 25 Star 298 Fork 54
Code Issues 15 Pull requests 3 Wiki Pulse Graphs Settings
Edit New issue

Operations Roadmap, Swift 3.0 & naming changes #343

Open danthorpe opened this issue 22 days ago · 9 comments

danthorpe commented 22 days ago • edited Owner

This ticket is to track and document the transition path of this project and the migration path for consumers of this project to Swift 3.0. There are a number of things which will happen.

Overall Migration Plan

1. Adopting a new name. As `Operation` has effectively been sherlocked by Swift 3.0, we need to pick a new name. The new name cannot have any other Swift 3.0 naming issues, such as `Task`, or maybe `Job`. So, currently settled on `Procedure`.
2. Refactor `Operation` to `Procedure` and `OperationQueue` to `ProcedureQueue`
3. Rename the project / module name. Rather than *Operations*, I am thinking *ProcedureKit*.
4. Move to an organisation account - again, ProcedureKit.

Labels: swift 3.0

Milestone: 4.0

Assignees: danthorpe

4 participants

Notifications