These are not the slides you’re looking for…

Operations on Steroids

App Frameworks

#WWDC15

Advanced NSOperations

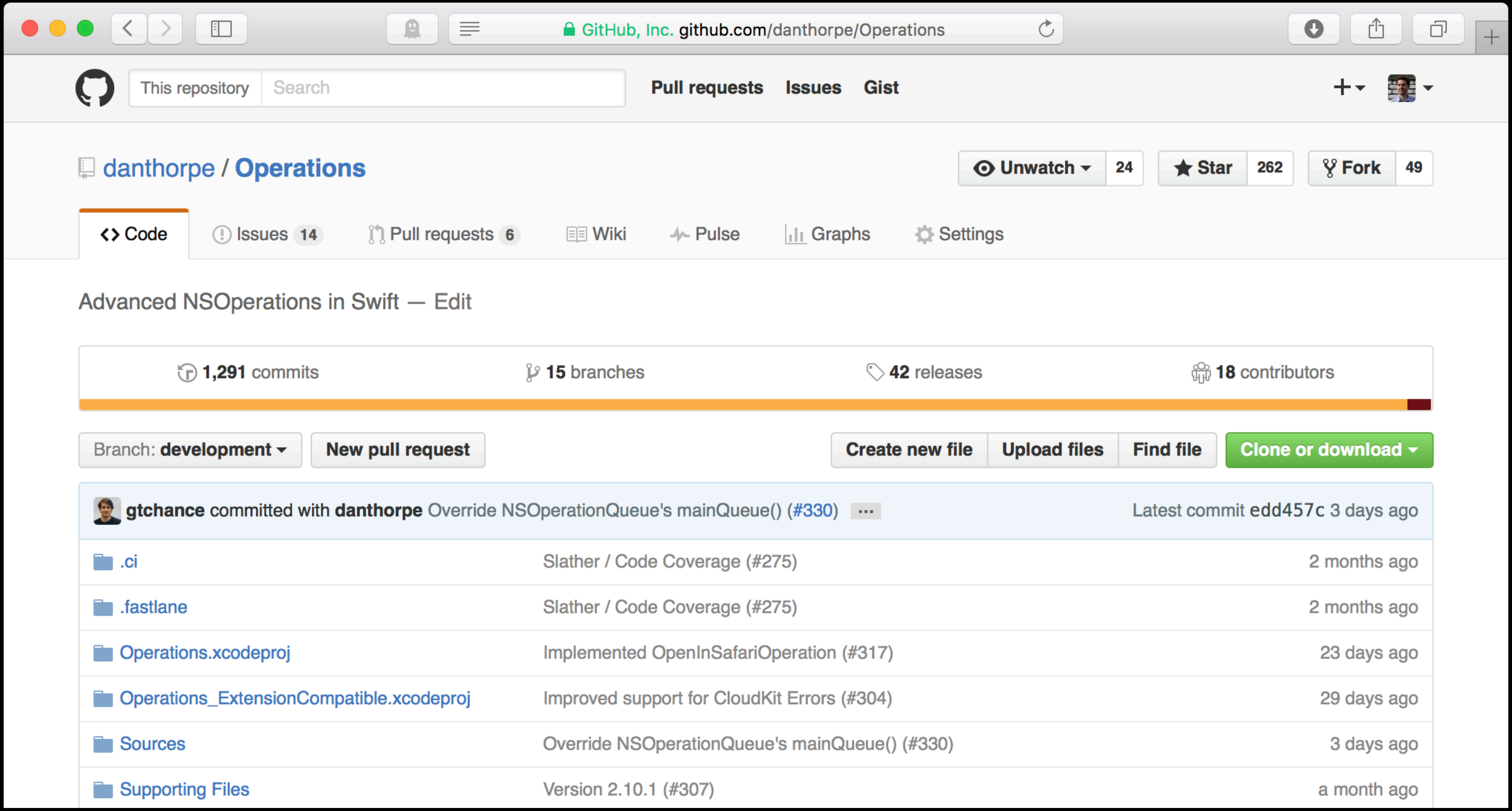
Exploring the WWDC app

Session 226

Philippe Hausler Foundation Engineer
Dave DeLong Frameworks Evangelist

https://github.com/danthorpe/

Operations


GitHub, Inc. github.com/danthorpe/Operations

This repository | Search | Pull requests | Issues | Gist

danthorpe / Operations

Unwatch 24 | Star 262 | Fork 49

Code | Issues 14 | Pull requests 6 | Wiki | Pulse | Graphs | Settings

Advanced NSOperations in Swift — Edit

1,291 commits | **15** branches | **42** releases | **18** contributors

Branch: **development** | New pull request | Create new file | Upload files | Find file | Clone or download

gtchance committed with **danthorpe** Override NSOperationQueue's mainQueue() (#330) … Latest commit edd457c 3 days ago

.ci	Slather / Code Coverage (#275)	2 months ago
.fastlane	Slather / Code Coverage (#275)	2 months ago
Operations.xcodeproj	Implemented OpenInSafariOperation (#317)	23 days ago
Operations_ExtensionCompatible.xcodeproj	Improved support for CloudKit Errors (#304)	29 days ago
Sources	Override NSOperationQueue's mainQueue() (#330)	3 days ago
Supporting Files	Version 2.10.1 (#307)	a month ago

Agenda

Recap

The Basics of Operations

Evolution

Beyond the Basics

Recap

NSOperationQueue

High-level abstraction of **dispatch_queue_t**

Supports cancellation

Variable width

NSOperation

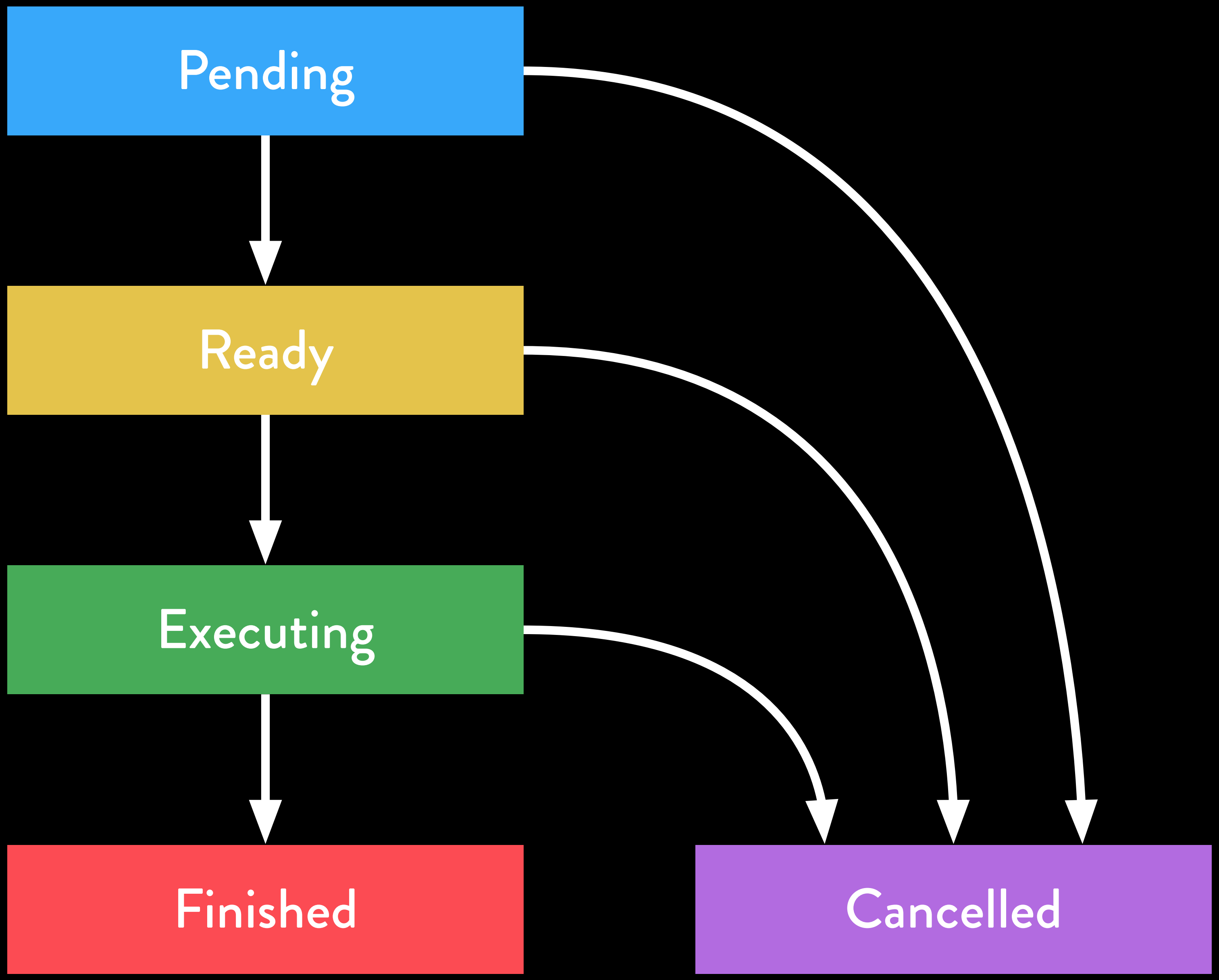
High-level abstraction of **dispatch_block_t**

Supports long running tasks

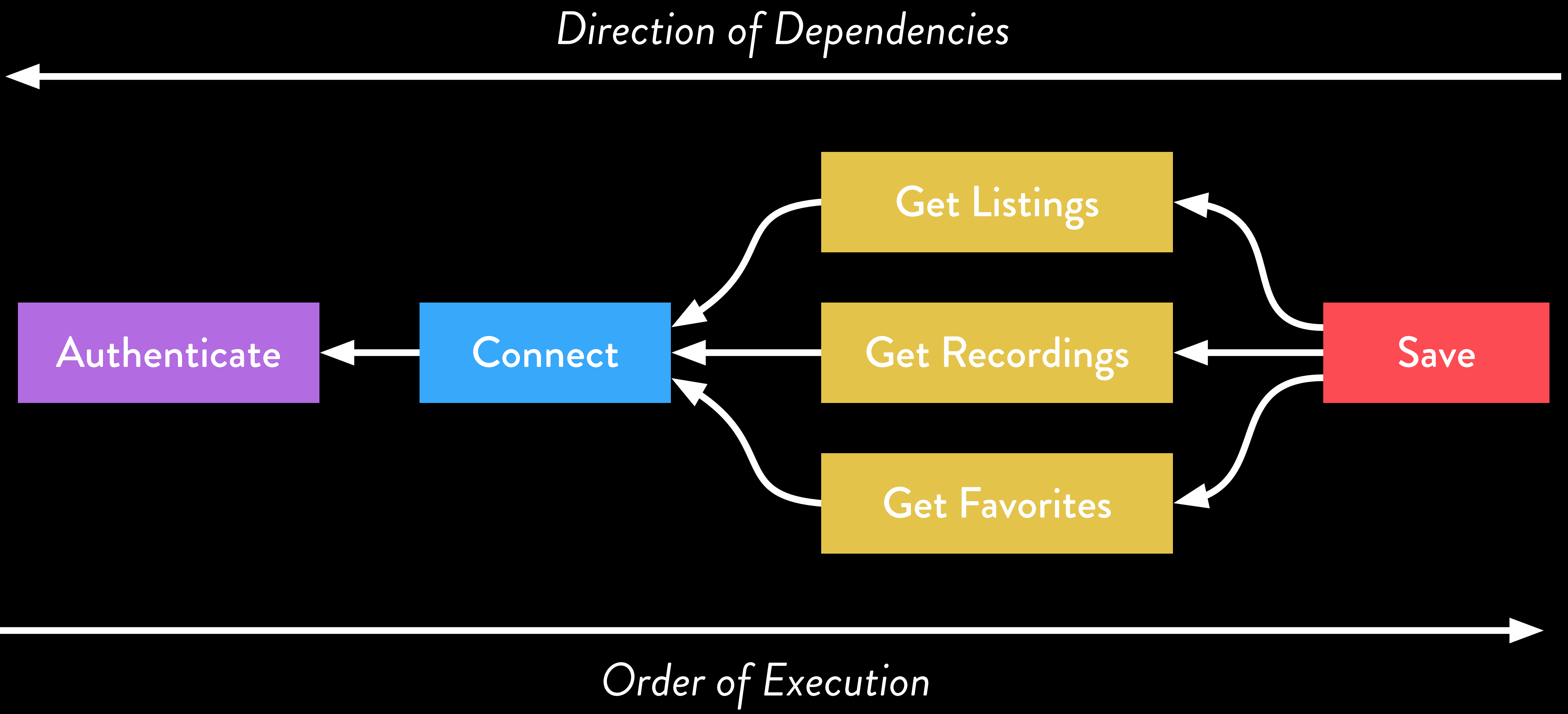
Supports object oriented design

Well defined lifecycle

NSOperation Lifecycle



Dependencies



Dependencies

Defines a strict ordering between operations

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

Works across operation queues

Abstraction

Operations abstract and isolate business logic

Encourages a decoupled and component-based architecture

Encourages code re-use

The Basics of Operations

Finished vs Failed

NSOperation instances always finish

No definition of success or failure

Operation supports the concept of finishing with errors, i.e. it failed

An operation which fails has still finished - dependencies beware!

Extending Readiness

Conditions are attached to an Operation

Conditions express the requirements needed to execute an operation

Conditions are evaluated to either succeed or fail with an error

Conditions can have their own dependencies

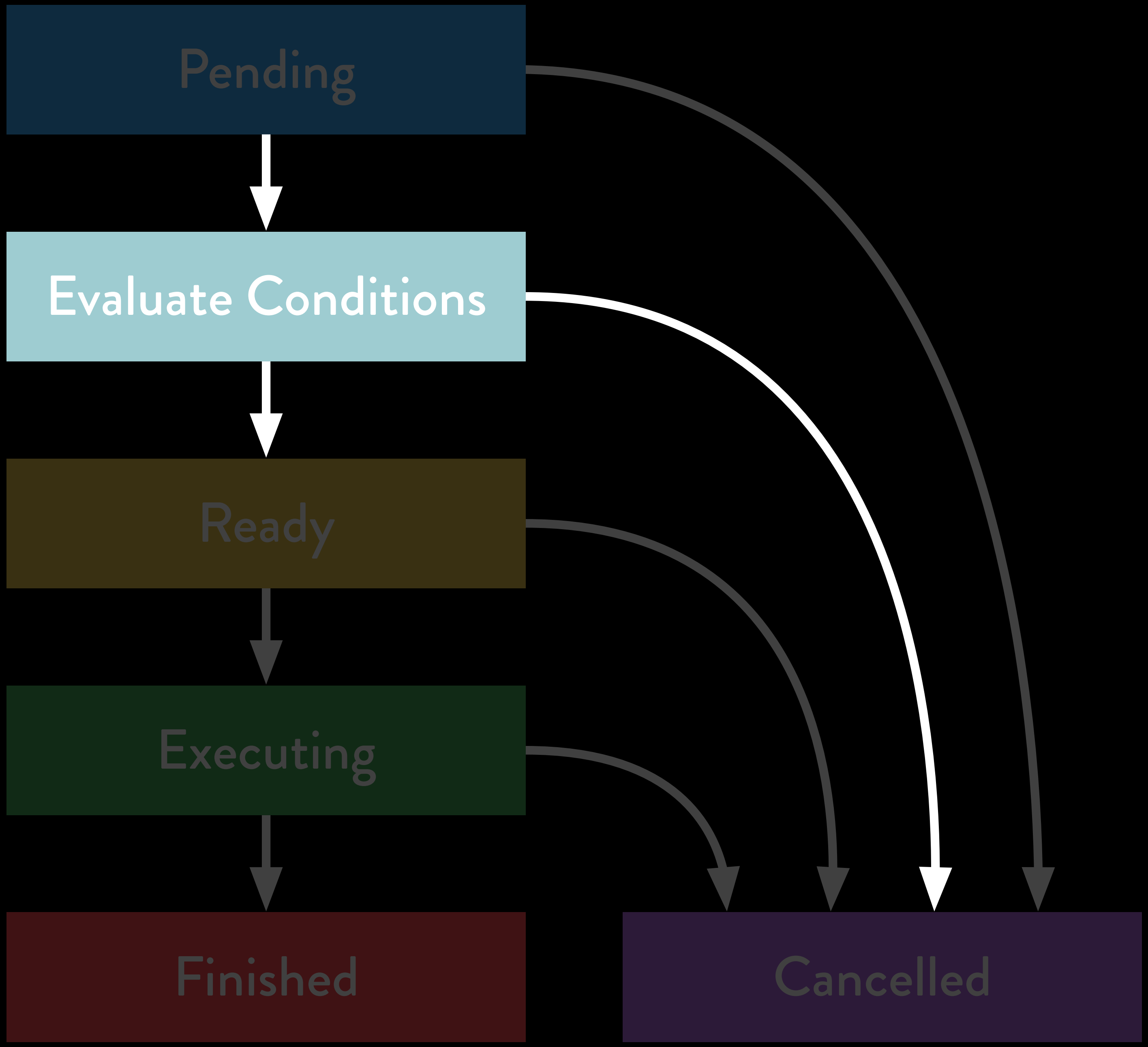
Lifecycle Events

Observers are attached to an Operation

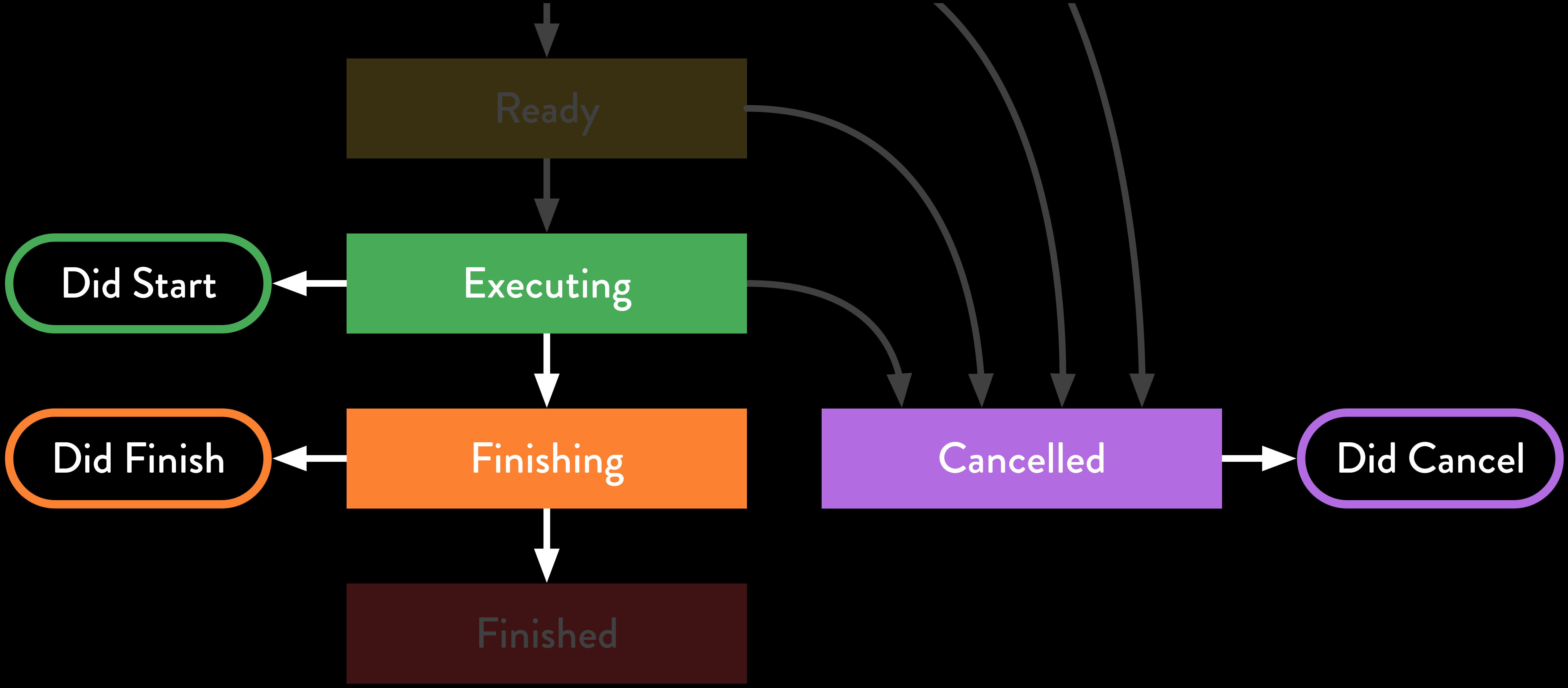
When the operation transitions between states it invokes the observers

Observer pattern decouples action/reaction logic

How *did* conditions work?



How *did* observers 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

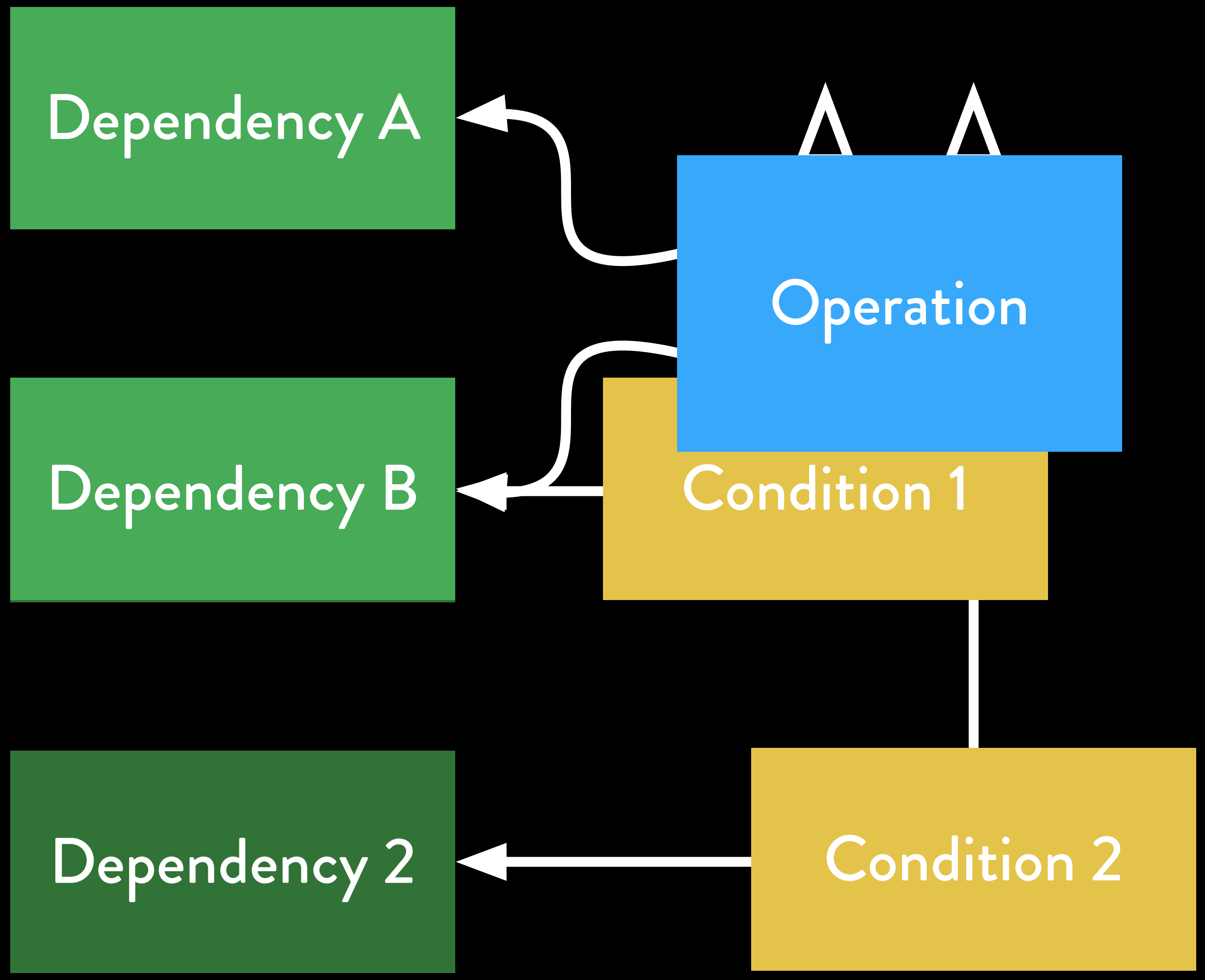
Conditions are now Operations

Add like a regular dependency

Avoids overriding isReady

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

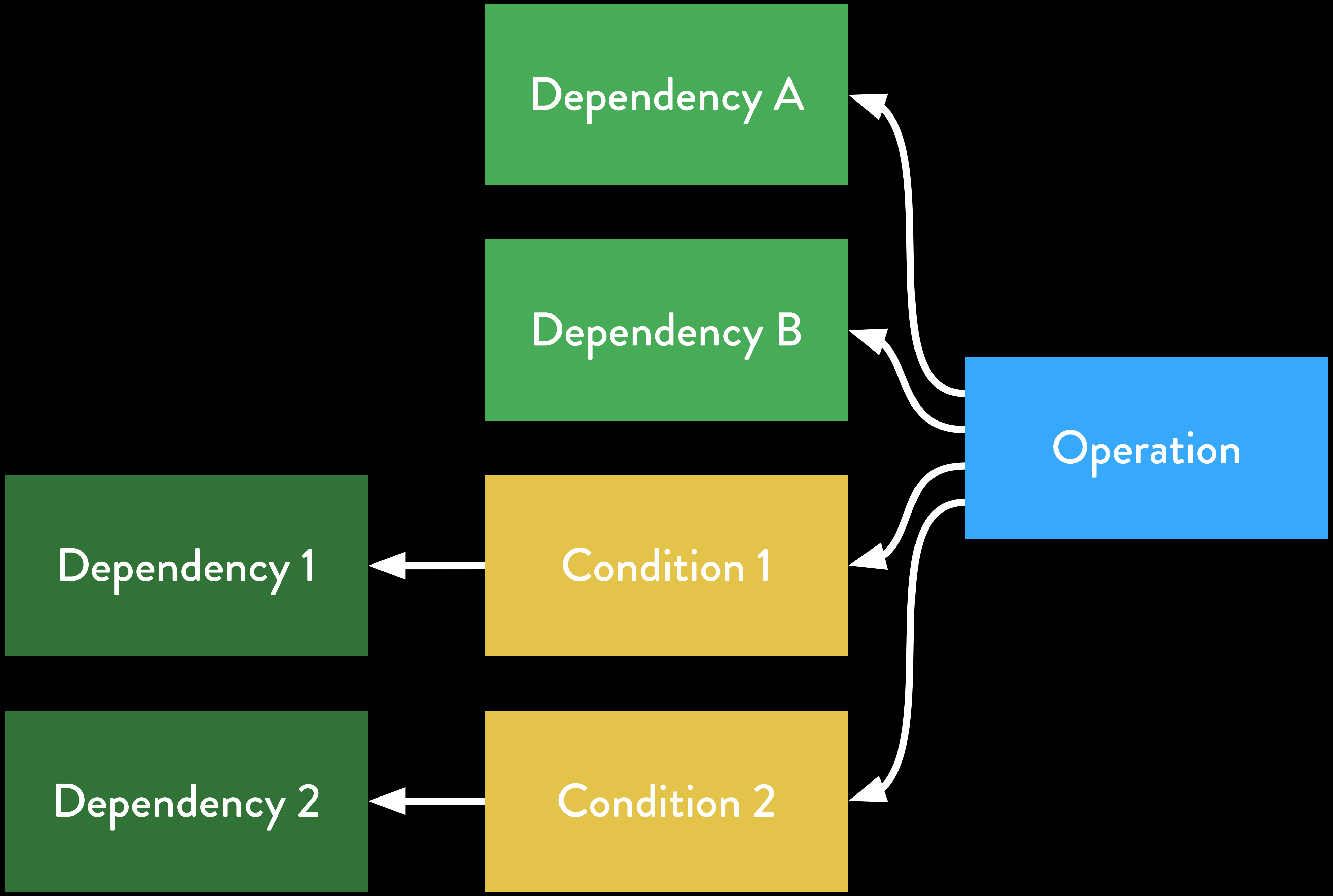
Conditions & Dependencies



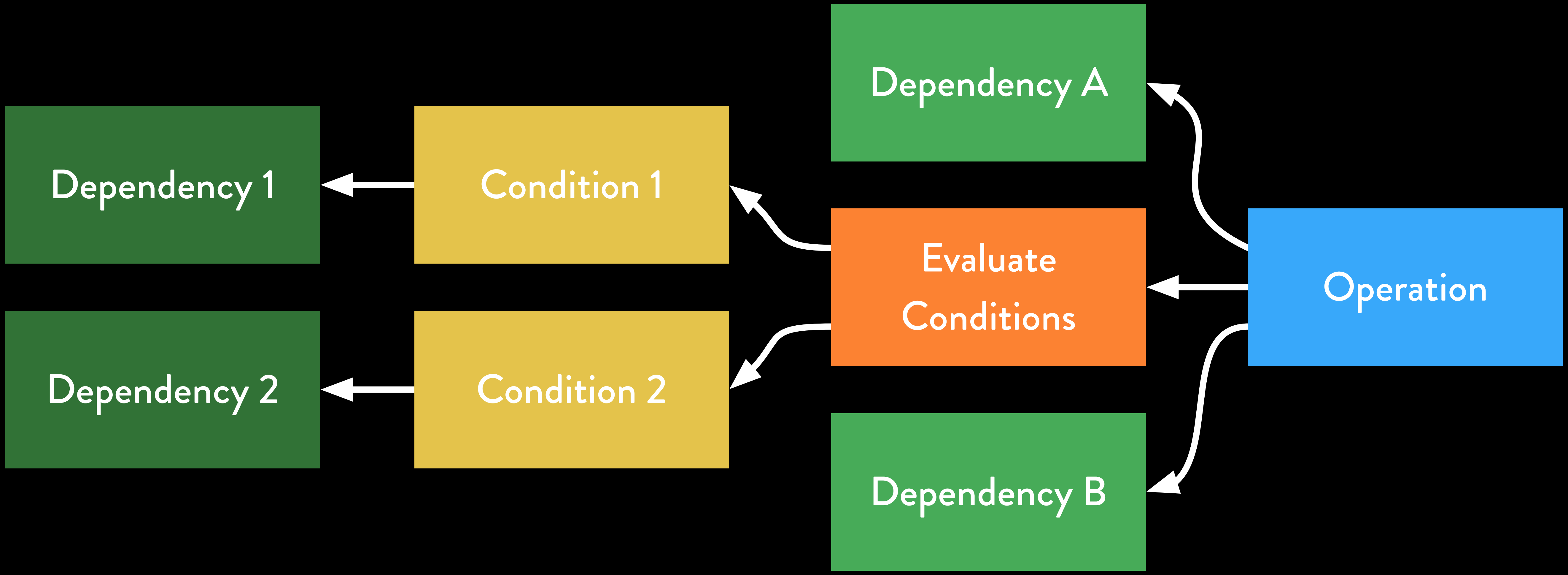
Conditions & Dependencies

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

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

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

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

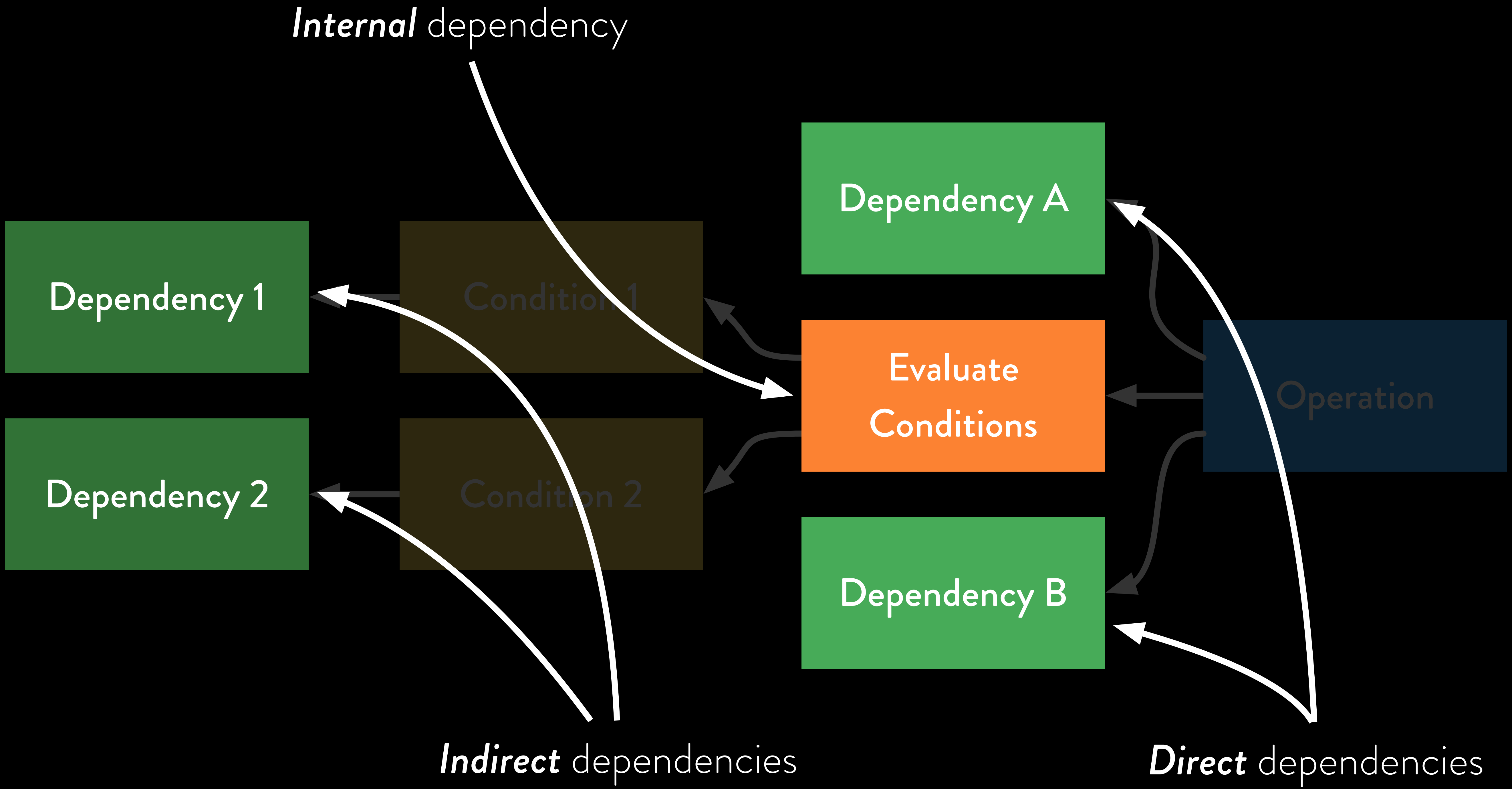
Scheduling


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

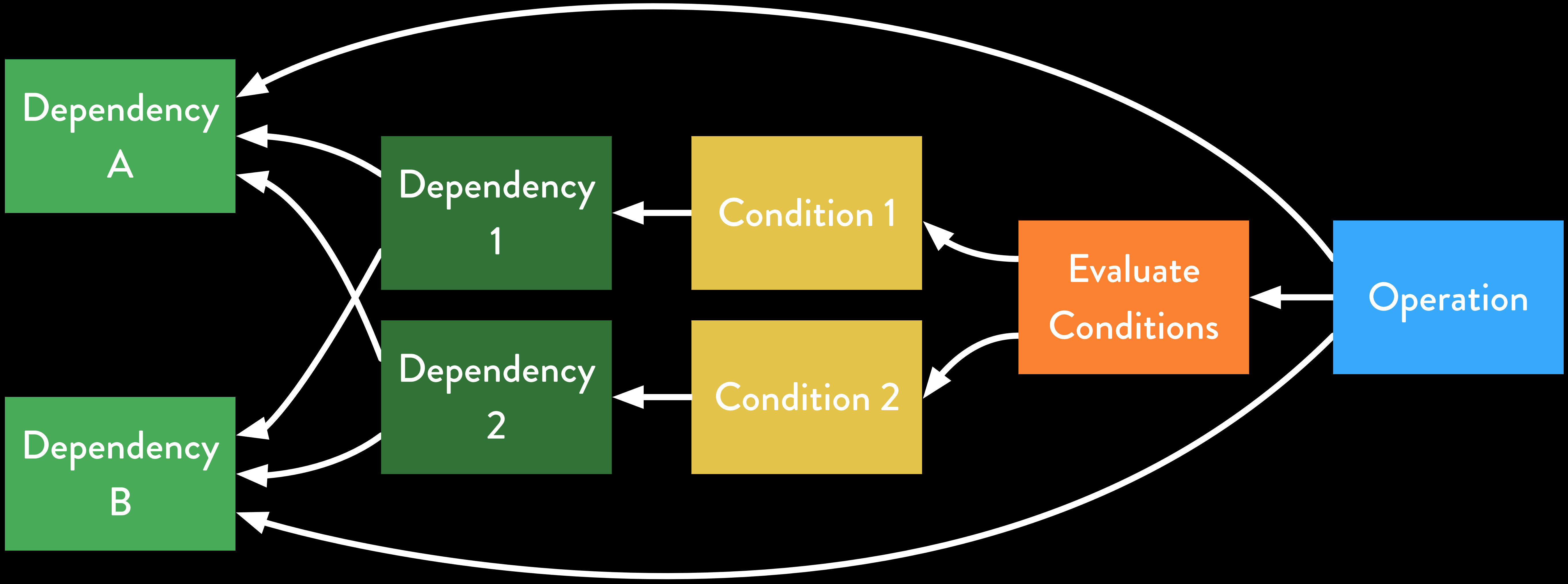
Scheduling


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

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

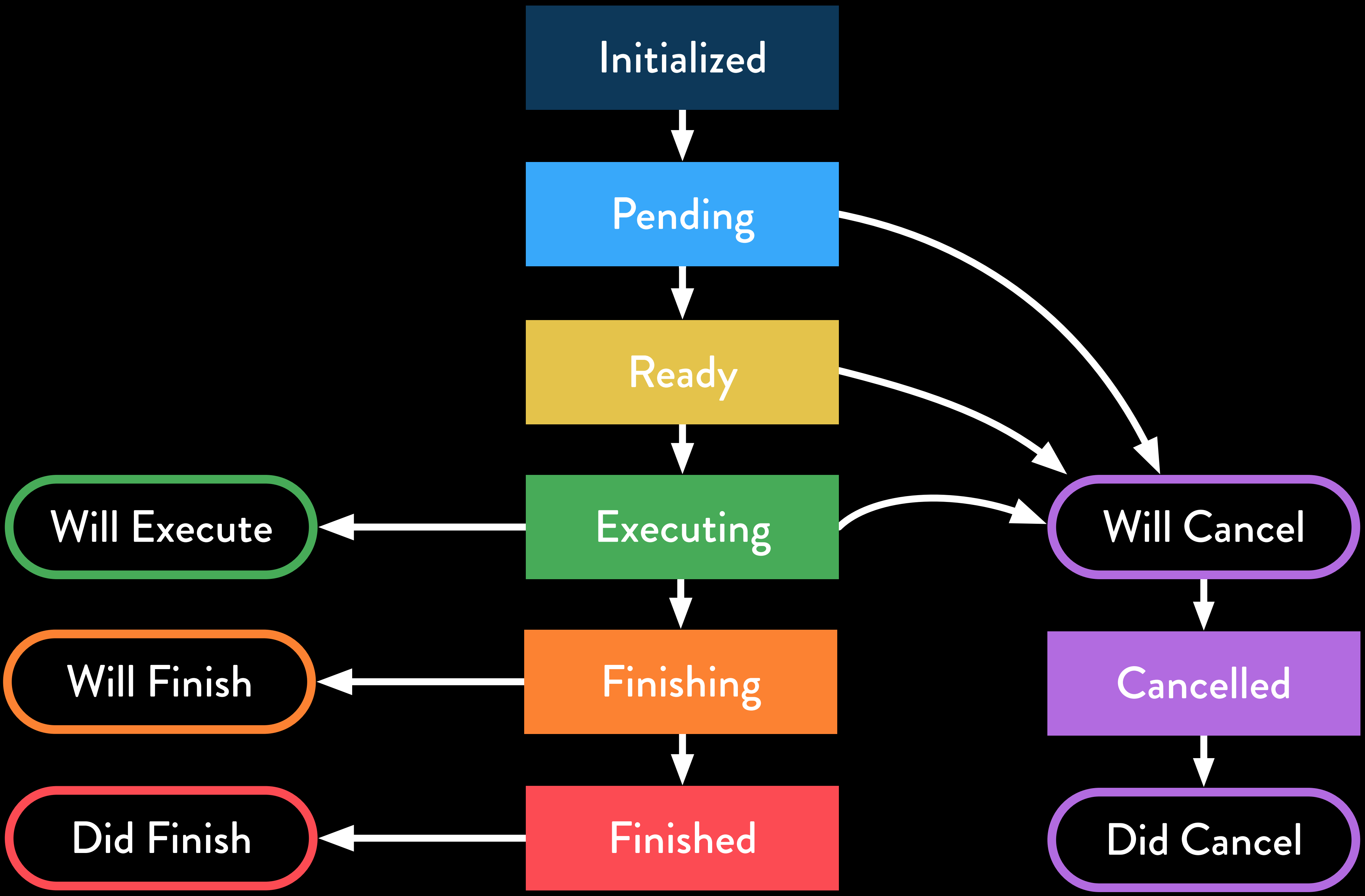
Triggering Observers

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
Finishing
Finished
Will Execute
Will Finish
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

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

Beyond the Basics

Result Injection

Often operations perform work, e.g. download a resource, open a file, crunch numbers

Typically this ***result*** is needed later as a ***requirement*** of another operation

Implicit dependency requirement between operations

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

Automatic Result Injection

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

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

```
operation.injectResultFromDependency(dependency)
```

Built in generic operations for map, filter and reduce.

Groups

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

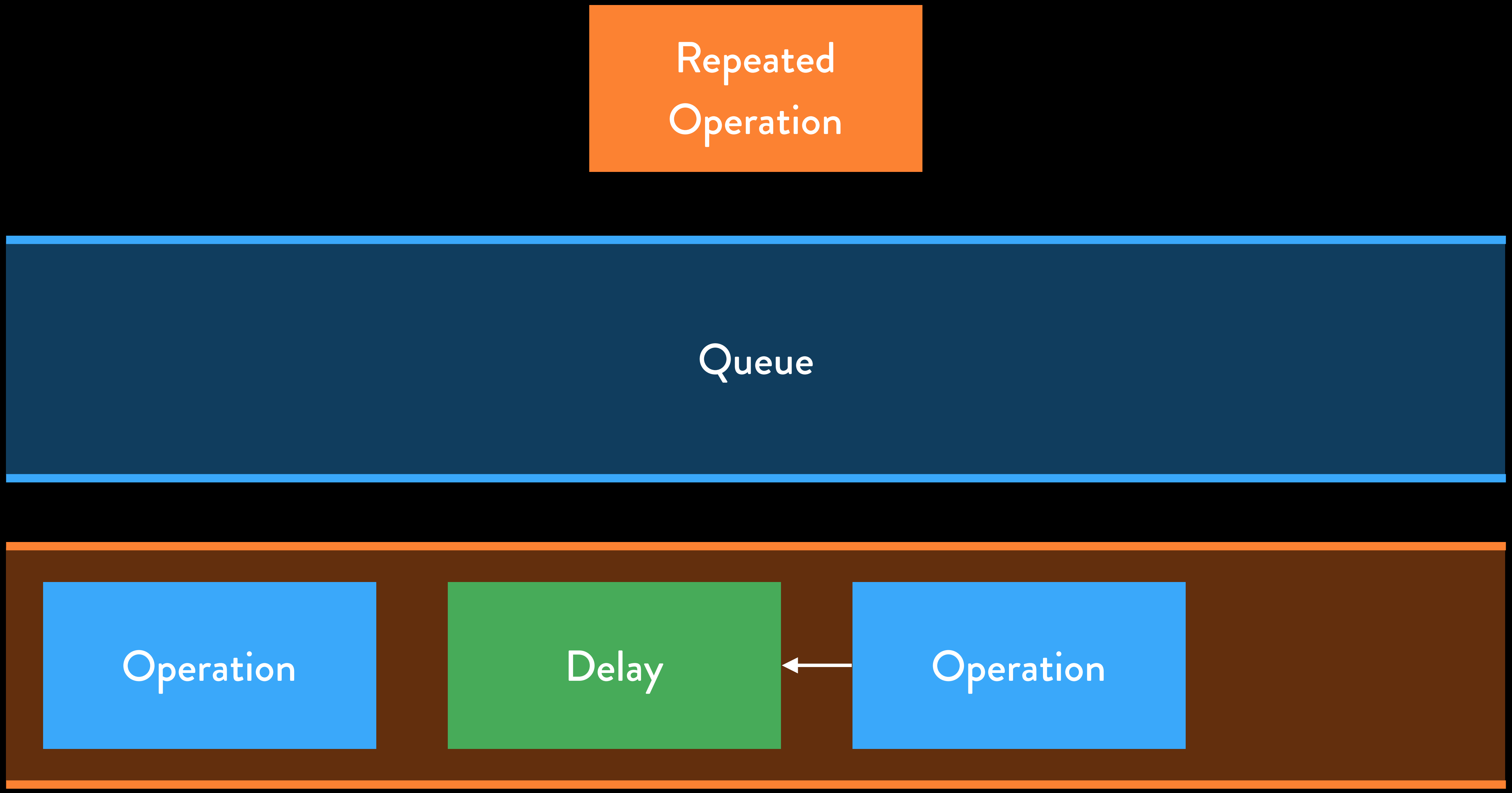
Generic GroupOperation subclass to support invoking the same operation type over and over

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

Further Abstraction

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

Application Features / Gatekeeper

Operations

Encapsulate work into a re-usable component

Combine components together to abstract logic

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!

Dan Thorpe

@danthorpe

danthorpe/Operations

danthorpe@me.com

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