DEPENDENCY INJECTION (DI) IN SWIFT

ILYA PUCHKA (@ILYAPUCHKA)
IOS DEVELOPER @HELLOFRESH

FUNCTIONAL OR OBJECT ORIENTED?

SOLID KISS DRY YAGNI RAP CQS DECORATOR FACADE ABSTRACT FACTORY STRATEGY ...

BI 🤝 SOLID

WHAT IS DEPENDENCY INJECTION?

IN SOFTWARE ENGINEERING, DEPENDENCY INJECTION IS A SOFTWARE DESIGN PATTERN THAT IMPLEMENTS INVERSION OF CONTROL FOR RESOLVING DEPENDENCIES.

– WIKIPEDIA

"DEPENDENCY INJECTION IS REALLY JUST PASSING IN AN INSTANCE VARIABLE.

– JAMES SHORE

VOODO MAGIC
SLOW FRAMEWORKS
ONLY FOR TESTING
OVERENGINEERING

- WHAT IS DEPENDENCY INJECTION?
- HOW TO DO IT?
- HOW NOT TO DO IT?

WHY DEPENDENCY INJECTION?

ABSTRACTIONS EVERYWHERE

LOOSE COUPLING

TEST

EXTEND AND REUSE

DEVELOP IN PARALLER

MAINTAIN

NOT JUST UNIT TESTS BUT LOOSE COUPLING

FIRST STEP - PASSING INSTANCE VARIABLES

SECOND STEP - ?

THIRD STEP - ?

PATTERNS

CUSTRUCTOR INJECTION

PROPERTY INJECTION

METHOD INJECTION

AMBIENT CONTEXT

CONSTRUCTOR INJECTION

```
class NSPersistentStore : NSObject {

    init(persistentStoreCoordinator root: NSPersistentStoreCoordinator?,
        configurationName name: String?,
        URL url: NSURL,
        options: [NSObject: AnyObject]?)

    var persistentStoreCoordinator: NSPersistentStoreCoordinator? { get }

}
```

EASY TO IMPLEMENT IMMUTABILITY

PROPERTY INJECTION

```
extension UIViewController {

    weak public var transitioningDelegate:
            UIViewControllerTransitioningDelegate?

}
```

LOCAL DEFAULT

FOREIGN DEFAULT

OPTIONALS
NOT IMMUTABLE
THREAD SAFETY

METHOD INJECTION

```
public protocol NSCoding {

    public func encodeWithCoder(aCoder: NSCoder)

}
```

AMBIENT CONTEXT

```
public class NSURLCache : NSObject {

    public class func setSharedURLCache(cache: NSURLCache)

    public class func sharedURLCache() -> NSURLCache

}
```

CROSS-CUTTING CONCERNS

- LOGGING
- ANALITYCS
- TIME/DATE
- ETC.

PROS:

- DOES NOT POLLUTE API
- DEPENDENCY ALWAYS AVAILABLE

CONS:

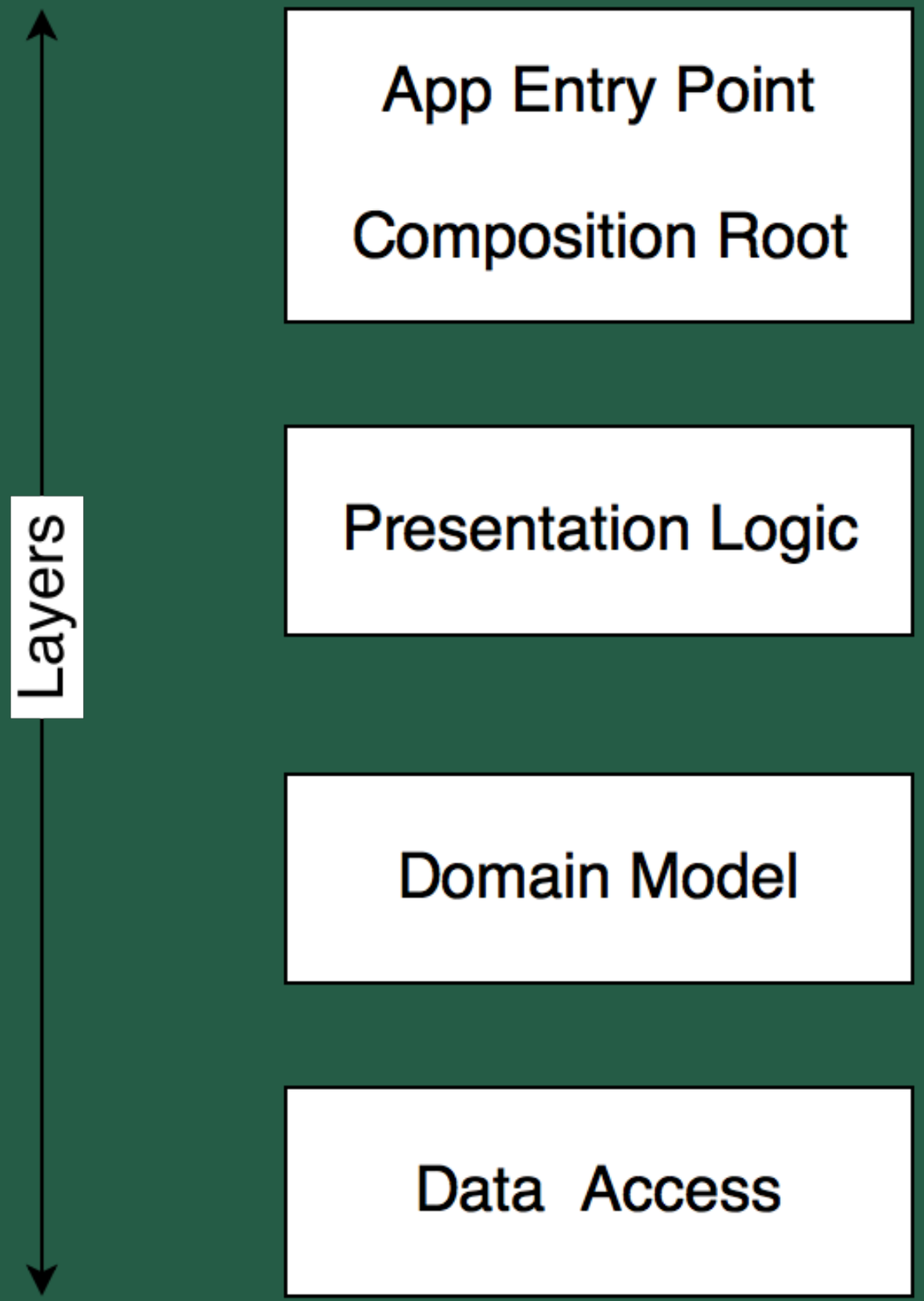
- IMPLICIT DEPENDENCY
- GLOBAL MUTABLE STATE

SEPARATION OF CONCERNS

- WHAT CONCRETE IMPLEMENTATIONS TO USE
- CONFIGURE DEPENDENCIES
- MANAGE DEPENDENCIES' LIFETIME

WHERE DEPENDENCIES ARE CREATED?

COMPOSITION ROOT


Layers
App Entry Point
Composition Root
Presentation Logic
Domain Model
Data Access

VIPER EXAMPLE

```
class AppDependencies {
    init() {
        configureDependencies()
    }

    func configureDependencies() {
        // Root Level Classes
        let coreDataStore = CoreDataStore()
        let clock = DeviceClock()
        let rootWireframe = RootWireframe()

        // List Module Classes
        let listPresenter = ListPresenter()
        let listDataManager = ListDataManager()
        let listInteractor = ListInteractor(dataManager: listDataManager, clock: clock)
        ...
        listInteractor.output = listPresenter
        listPresenter.listInteractor = listInteractor
        listPresenter.listWireframe = listWireframe
        listWireframe.addWireframe = addWireframe
        ...
    }
}
```

VIPER EXAMPLE

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    let appDependencies = AppDependencies()

    func application(
        application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject : AnyObject]?) -> Bool {

        appDependencies.installRootViewControllerIntoWindow(window!)

        return true
    }
}
```

THE BIGGEST CHALLANGE OF PROPERLY IMPLEMENTING DI IS GETTING ALL CLASSES WITH DEPENDENCIES MOVED TO COMPOSITION ROOT

– MARK SEEMAN

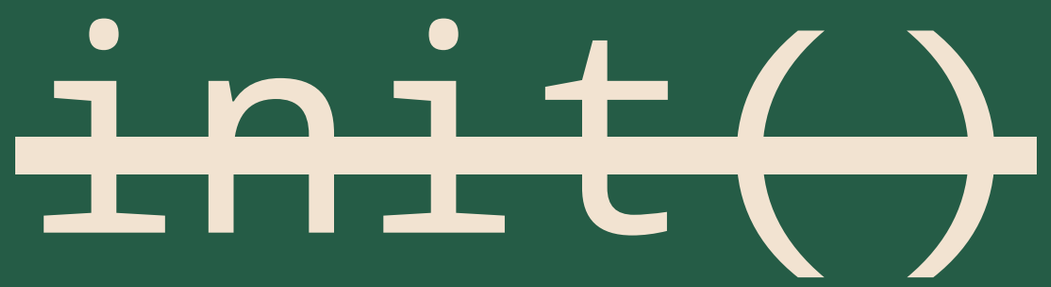
ANTI-
PATTERNS

CONTROL FREAK

```
class RecipesService {

    let repository: RecipesRepository

    init() {
        self.repository = CoreDataRecipesRepository()
    }
}
```


init()

STABLE

VOLATILE

VOLATILE DEPENDENCIES

- DEPENDENCY REQUIRES ENVIRONMENT CONFIGURATION (DATA BASE. NETWORKING. FILE SYSTEM)
- NONDETERMENISTIC BEHAVIOR (DATES. CRYPTOGRAPHY)
- EXPECTED TO BE REPLACED
- DEPENDENCY IS STILL IN DEVELOPMENT

VOLATILE DEPENDENCIES

DISABLE LOOSE COUPLING

BASTARD INJECTION

```
class RecipesService {

    let repository: RecipesRepository

    init(repository: RecipesRepository = CoreDataRecipesRepository()) {
        self.repository = repository
    }
}
```

FOREIGN DEFAULT

SERVICE LOCATOR

```
let locator = ServiceLocator.sharedInstance

locator.register( { CoreDataRecipesRepository() },
                  forType: RecipesRepository.self)

class RecipesService {

    let repository: RecipesRepository

    init() {
        let locator = ServiceLocator.sharedInstance
        self.repository = locator.resolve(RecipesRepository.self)
    }
}
```

PROS:

- EXTENSIBILITY
- TESTABILITY
- PARALLEL DEVELOPMENT
- SEPARATION OF CONCERNS

CONS:

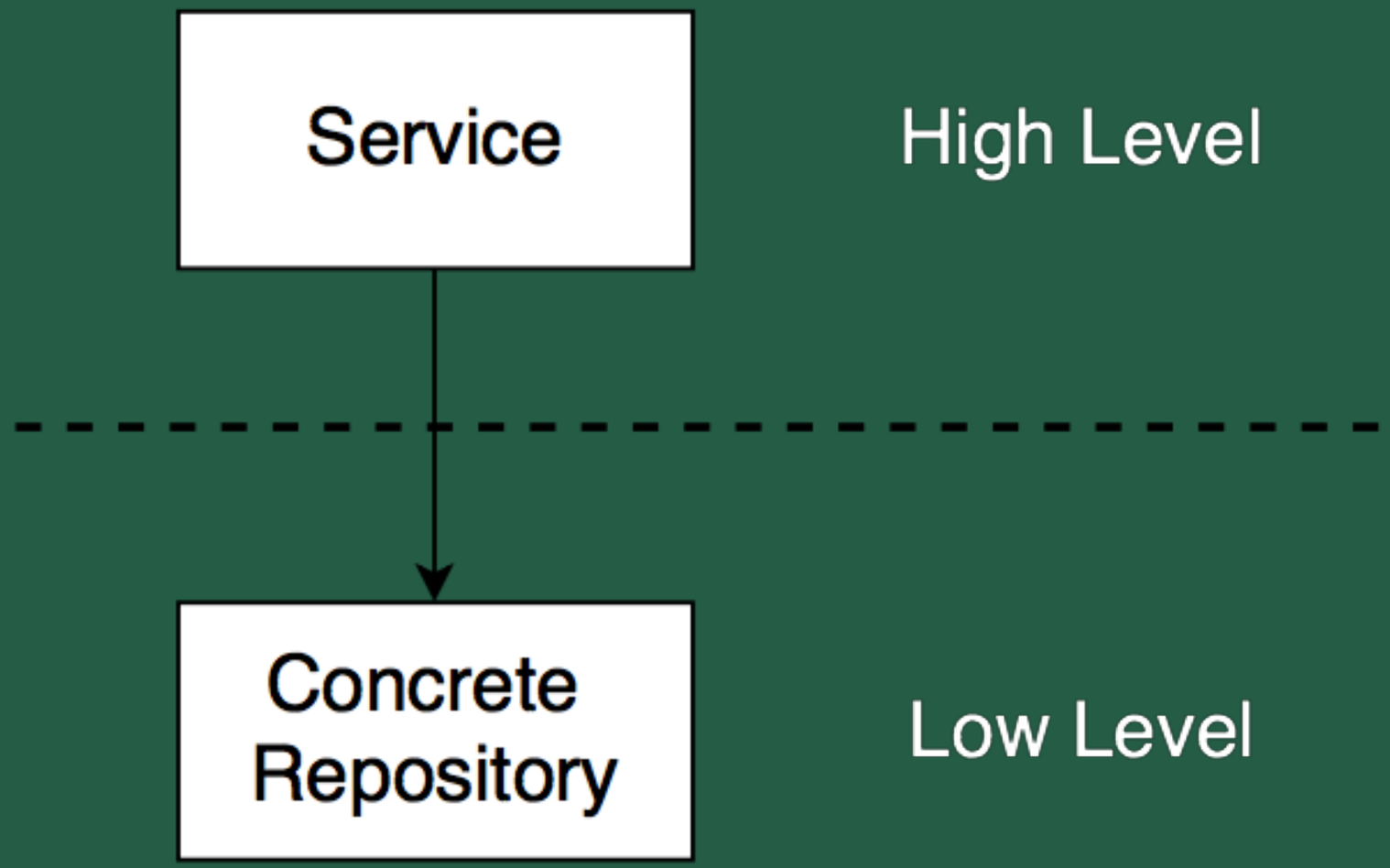
- IMPLICIT DEPENDENCIES
- HIDDEN COMPLEXITY
- TIGHT COUPLING
- NOT REUSABLE
- LESS MAINTAINABLE

- DI ENABLES LOOSE COUPLING
- 4 PATTERNS. PREFER CONSTRUCTOR INJECTION
- USE LOCAL DEFAULTS, NOT FOREIGN
- INJECT VOLATILE DEPENDENCIES. NOT STABLE
- AVOID ANTI-PATTERNS

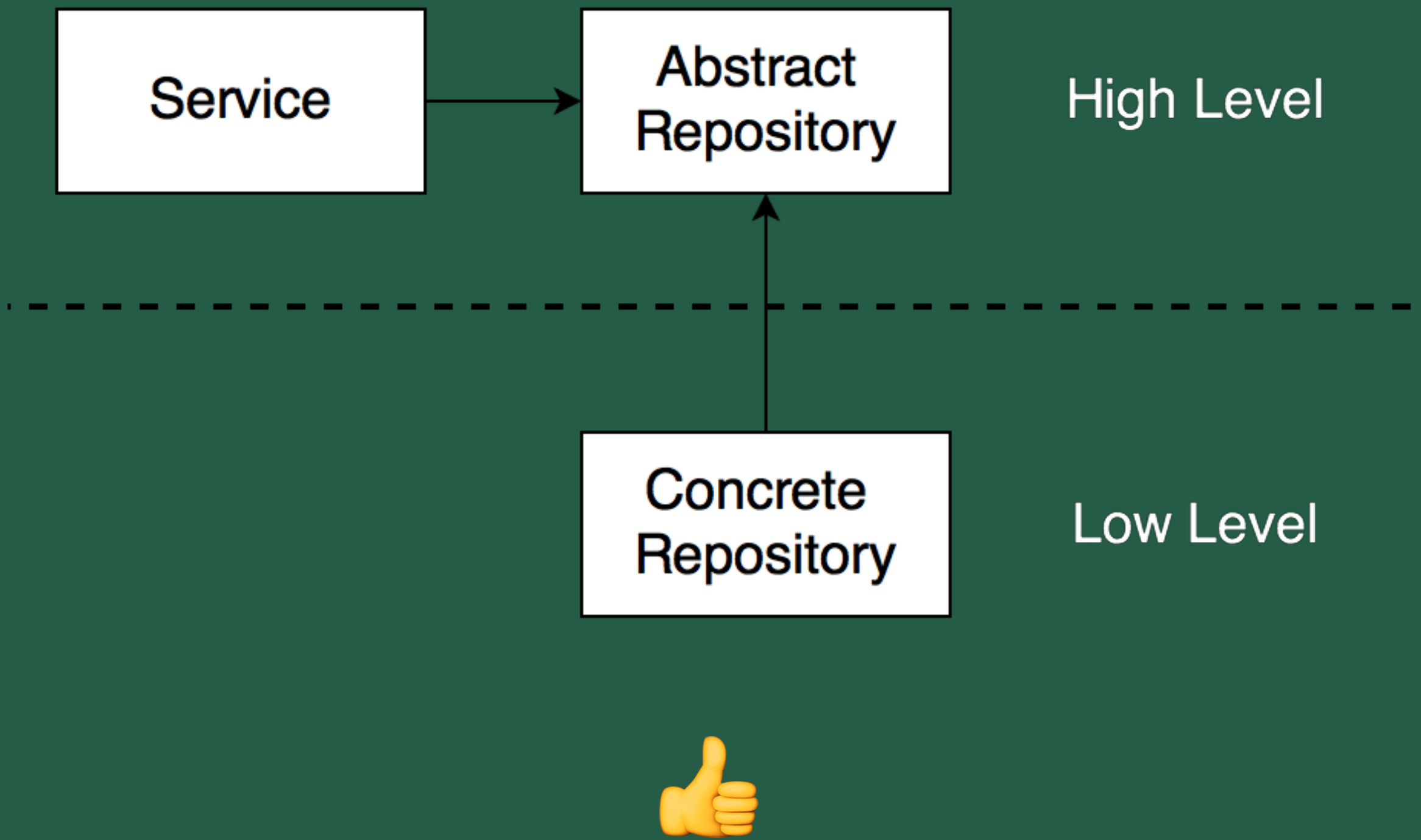
EXPLICIT DEPENDENCIES COMPOSITION ROOT

SECOND STEP – ABSTRACTIONS

DEPENDENCY INVERSION PRINCIPLE (DIP)


Service
High Level
Concrete
Repository
Low Level

👎


Service
Abstract Repository
High Level
Concrete Repository
Low Level

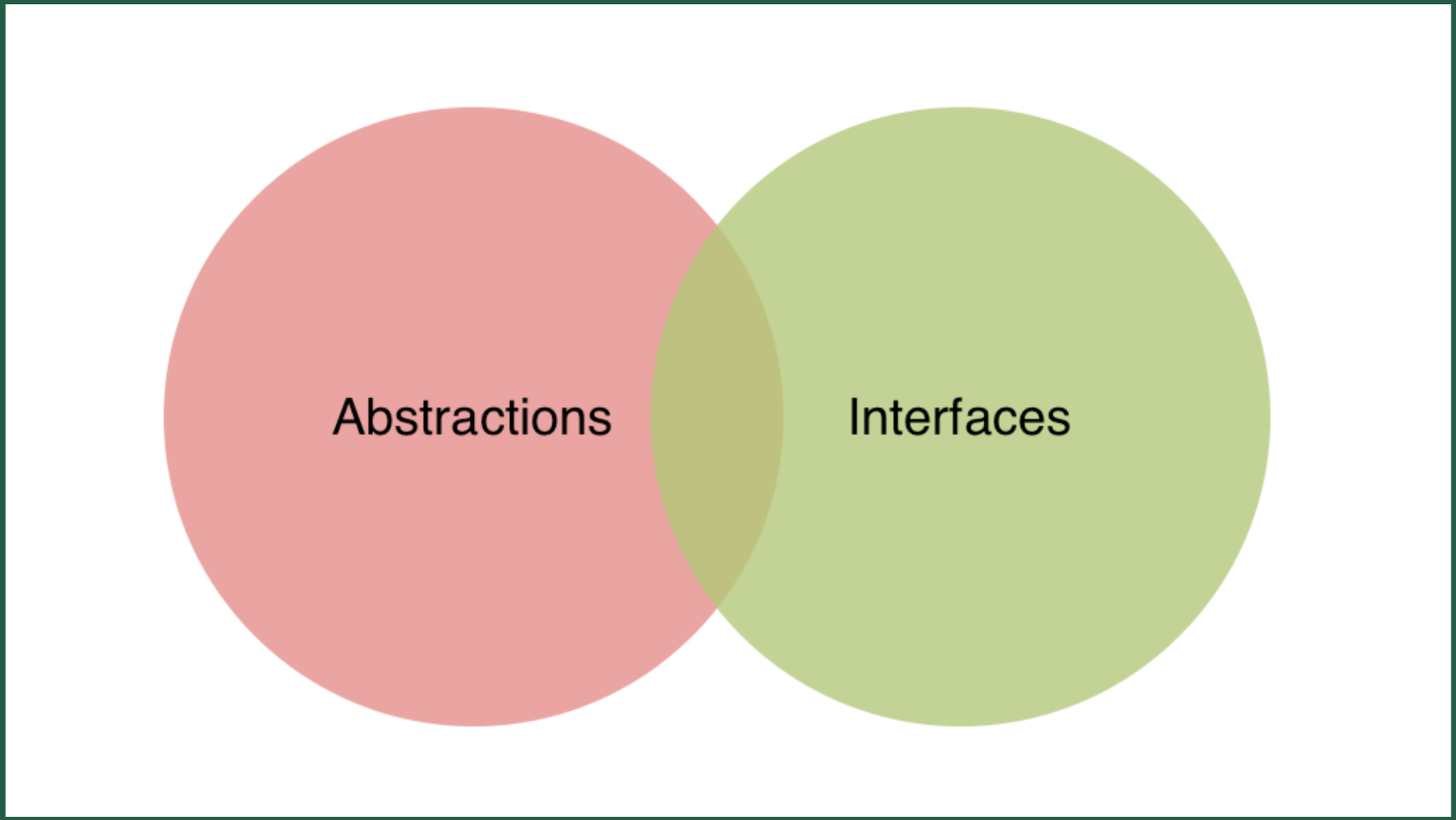
DI = DI PATTERNS + DIP

PROGRAM TO AN INTERFACE, NOT AN IMPLEMENTATION

– DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE

PROGRAM TO AN INTERFACE ABSTRACTION

INTERFACES ARE NOT ABSTRACTIONS [1]



[1] HTTP://BLOG.PLOEH.DK/2010/12/02/INTERFACESARENOTABSTRACTIONS/

DI = DI PATTERNS + DIP

THIRD STEP - INVERSION OF CONTROL

DI CONTAINERS

TYPHOON

DIP

TYPHOON

HTTP://TYPHOONFRAMEWORK.ORG

- A LOT OF FEATURES
- GOOD DOCS
- WELL MAINTAINED
- CONTINUOUSLY IMPROVED

```
public class APIClientAssembly: TyphoonAssembly {

    public dynamic func apiClient() -> AnyObject {
        ...
    }

    public dynamic func session() -> AnyObject {
        ...
    }

    public dynamic func logger() -> AnyObject {
        ...
    }

}
```

```
public dynamic func apiClient() -> AnyObject {
    return TyphoonDefinition.withClass(APIClientImp.self) { definition in

        definition.useInitializer(#selector(APIClientImp.init(session:))) {
            initializer in

            initializer.injectParameterWith(self.session())
        }

        definition.injectProperty("logger", with: self.logger())
    }
}
```

```
public dynamic func session() -> AnyObject {
    return TyphoonDefinition.withClass(NSURLSession.self) { definition in
        definition.useInitializer(#selector(NSURLSession.sharedSession))
    }
}
```

```
public dynamic func logger() -> AnyObject {
    return TyphoonDefinition.withClass(ConsoleLogger.self) { definition in
        definition.scope = .Singleton
    }
}
```

```
let assembly = APIClientAssembly().activate()
let apiClient = assembly.apiClient() as! APIClient
```

TYPHOON + SWIFT 🤔

- REQUIRES TO SUBCLASS NSOBJECT AND DEFINE PROTOCOLS WITH `@objc`
- METHODS CALLED DURING INJECTION SHOULD BE `dynamic`
- REQUIRES TYPE CASTING
- NOT ALL FEATURES WORK IN SWIFT
- TOO WORDY API FOR SWIFT

DIP

https://github.com/alisoftware/dip

- Pure Swift API
- Cross-platform
- Type-safe
- Small code base

REGISTER

```
let container = DependencyContainer()

container.register {
    try APIClientImp(
        session: container.resolve()
    ) as APIClient
}
.resolveDependencies { container, client in
    client.logger = try container.resolve()
}

container.register { NSURLSession.sharedSession() as NetworkSession }
container.register(.Singleton) { ConsoleLogger() as Logger }
```

RESOLVE

```
let apiClient = try! container.resolve() as APIClient
```

AUTO-WIRING

```
class APIClientImp: APIClient {

    private let _logger = Injected<Logger>()

    var logger: Logger? { return _logger.value }

}
```

AUTO-WIRING

```
class APIClientImp: APIClient {
    init(session: NetworkSession) { ... }
}
```

```
container.register {
    APIClientImp(session: $0) as APIClient
}
```

	Typhoon	Dip
Constructor, property, method injection	✔	✔
Lifecycle management	✔	✔
Circular dependencies	✔	✔
Runtime arguments	✔	✔
Named definitions	✔	✔
Storyboards integration	✔	✔
Auto-wiring	✔	✔
Thread safety	✘	✔
Interception	✔	✘
Infrastructure	✔	✘

WHY SHOULD I BOTHER?

- EASY INTEGRATION WITH STORYBOARDS
- MANAGE COMPONENTS LIFECYCLE
- CAN SIMPLIFY CONFIGURATIONS
- ALLOW INTERCEPTION (IN TYPHOON USING NSPROXY)
- PROVIDES ADDITIONAL FEATURES

DI ≠ DI CONTAINER

DEPENDENCY INJECTION IS A MEANS TO AN END

LINKS

- "DEPENDENCY INJECTION IN .NET" MARK SEEMAN
  - MARK SEEMAN'S BLOG
- OBJC.IO ISSUE 15: TESTING, DEPENDENCY INJECTION, BY JON REID
  - "DIP IN THE WILD"
- NON-DI CODE == SPAGHETTI CODE?

THANK YOU!

@ILYAPUCHKA

HTTP://ILYA.PUCHKA.ME