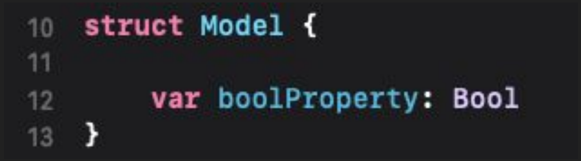
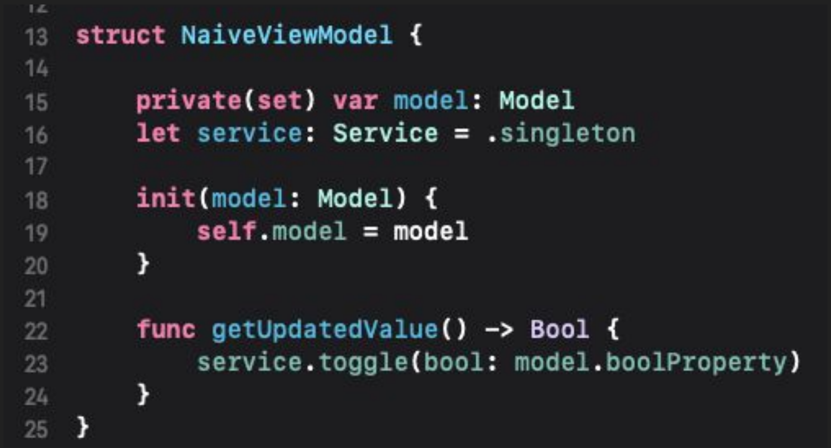
Dependency Injection 💉 & Service Locators 🔍

a.k.a.
how to manage dependencies and ensure testability

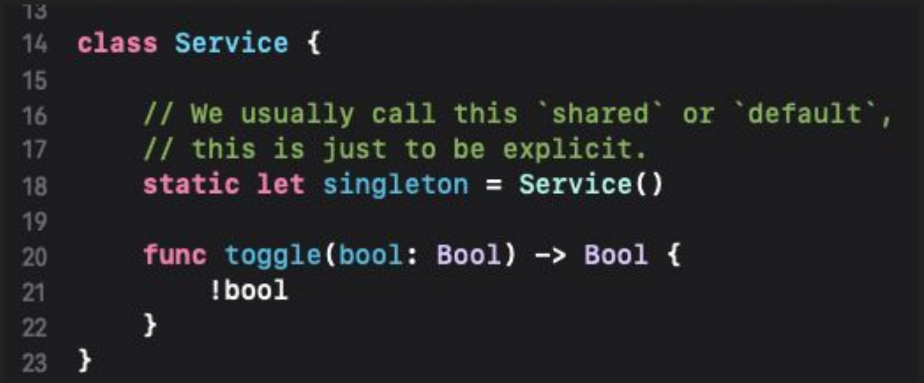
A simple MVVM codebase that has a Service



```swift
struct Model {

    var boolProperty: Bool
}
```



```swift
struct NaiveViewModel {

    private(set) var model: Model
    let service: Service = .singleton

    init(model: Model) {
        self.model = model
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```



```swift
class Service {

    // We usually call this `shared` or `default`,
    // this is just to be explicit.
    static let singleton = Service()

    func toggle(bool: Bool) -> Bool {
        !bool
    }
}
```

Single responsibility
Open–closed
Liskov substitution
Interface segregation
Dependency inversion

Dependency Inversion Principle:

1. High-level modules should not import anything from low-level modules.
   Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details.
   Details (concrete implementations) should depend on abstractions.

Dependency Inversion Principle:

1. High-level modules should not import anything from low-level modules.
   Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details.
   Details (concrete implementations) should depend on abstractions.

What does this mean? Let’s break it down.

Dependency Inversion Principle:

1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g. ~~interfaces~~ protocols).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

What does this mean? Let's break it down.

If we have an object (`class Object`) that is necessary for some piece of code to work, we should replace it with a protocol representing it (`protocol ObjectRepresentable`), and have the object conform to that protocol (`class Object: ObjectRepresentable`).

That way, we can more easily inject (foreshadowing) a substitute in its place, without worrying about the implementation details of the original object.

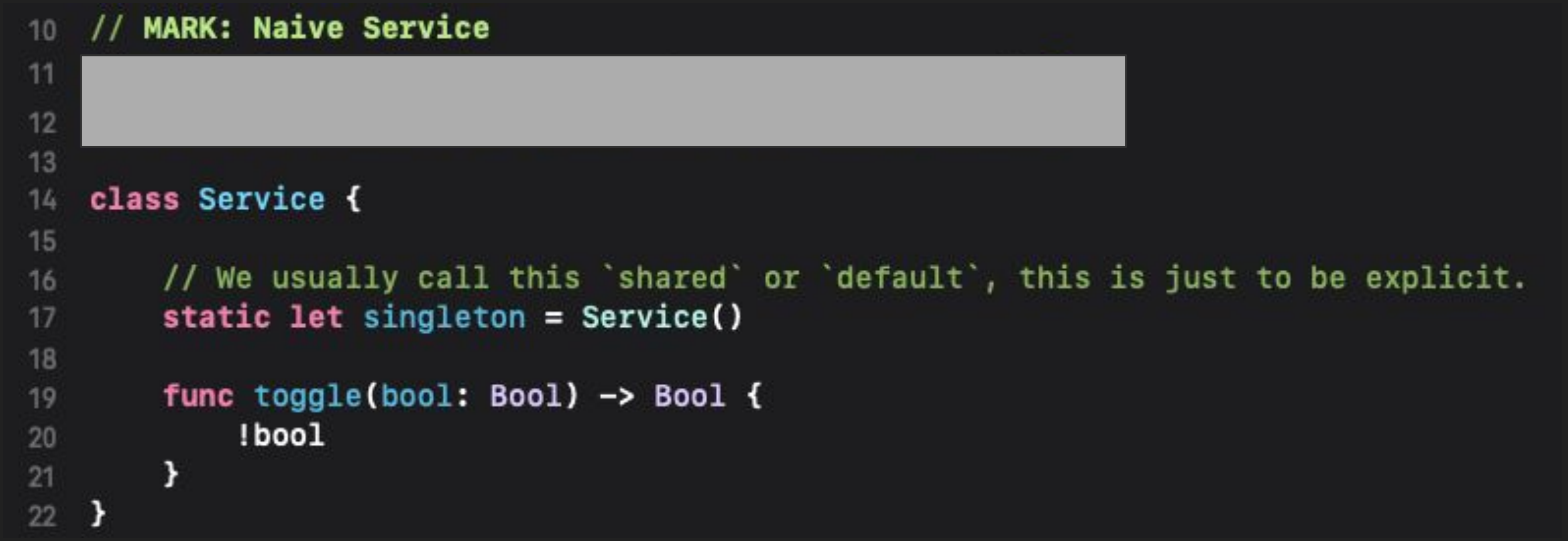
Dependency Injection

A.k.a.

don’t hide your dependencies in the details

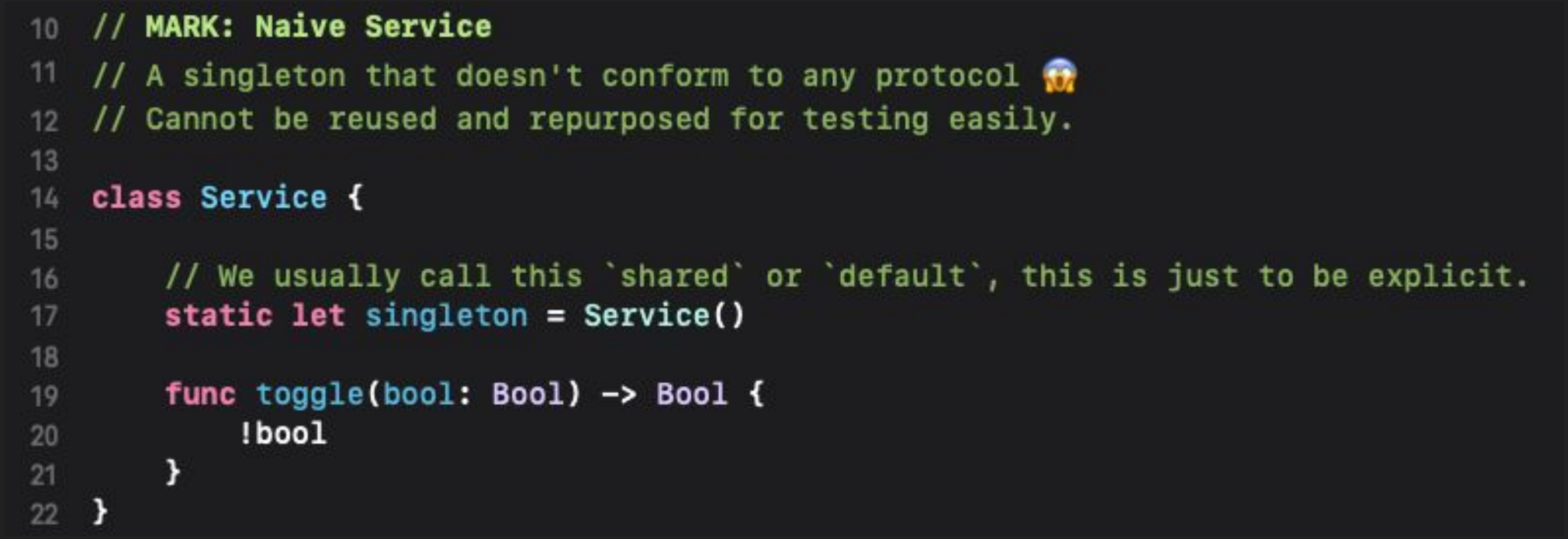
Service DI

Service



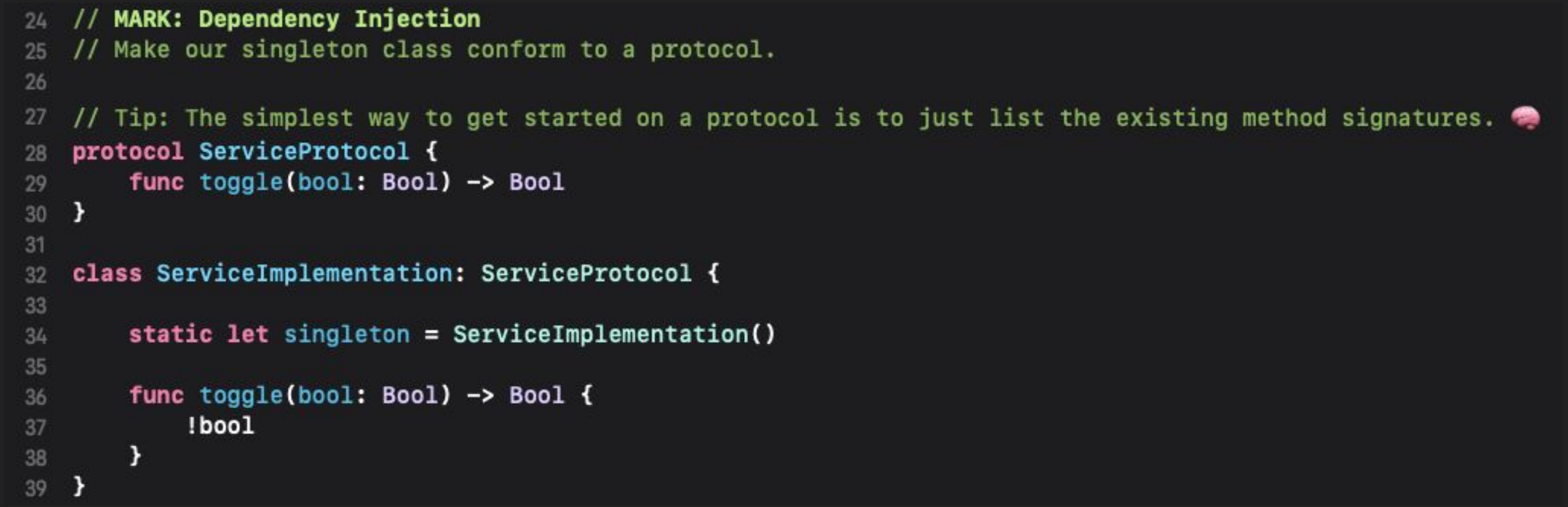
```swift
// MARK: Naive Service
[redacted]

class Service {

    // We usually call this `shared` or `default`, this is just to be explicit.
    static let singleton = Service()

    func toggle(bool: Bool) -> Bool {
        !bool
    }
}
```

Service



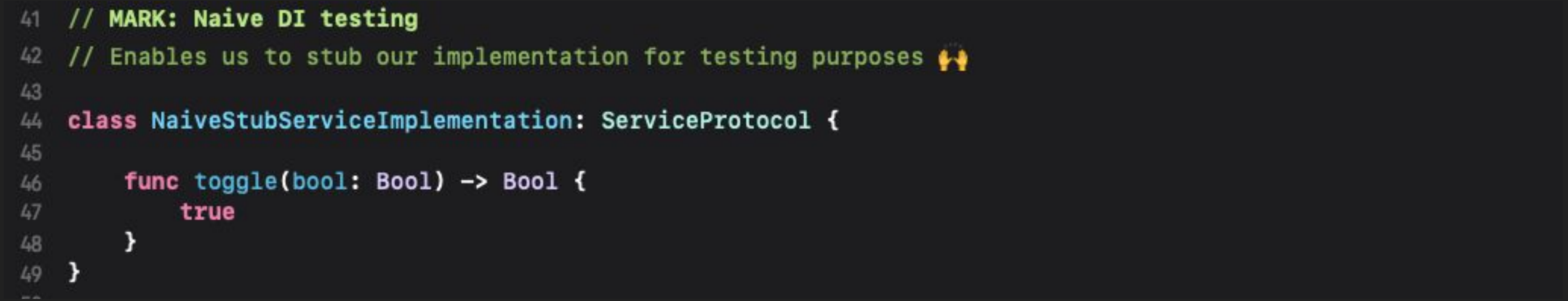
```swift
// MARK: Naive Service
// A singleton that doesn't conform to any protocol 😱
// Cannot be reused and repurposed for testing easily.

class Service {

    // We usually call this `shared` or `default`, this is just to be explicit.
    static let singleton = Service()

    func toggle(bool: Bool) -> Bool {
        !bool
    }
}
```

Service



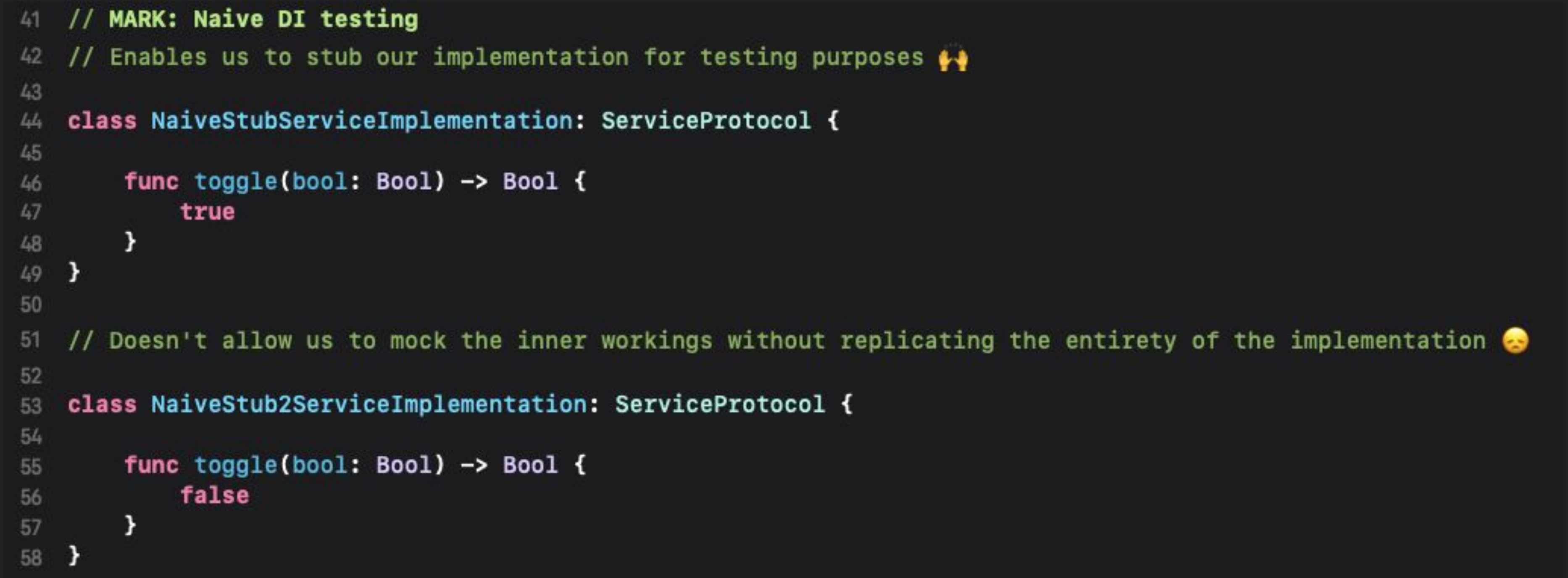
```swift
// MARK: Dependency Injection
// Make our singleton class conform to a protocol.

// Tip: The simplest way to get started on a protocol is to just list the existing method signatures. 🧠
protocol ServiceProtocol {
    func toggle(bool: Bool) -> Bool
}

class ServiceImplementation: ServiceProtocol {

    static let singleton = ServiceImplementation()

    func toggle(bool: Bool) -> Bool {
        !bool
    }
}
```

Service Stubbing



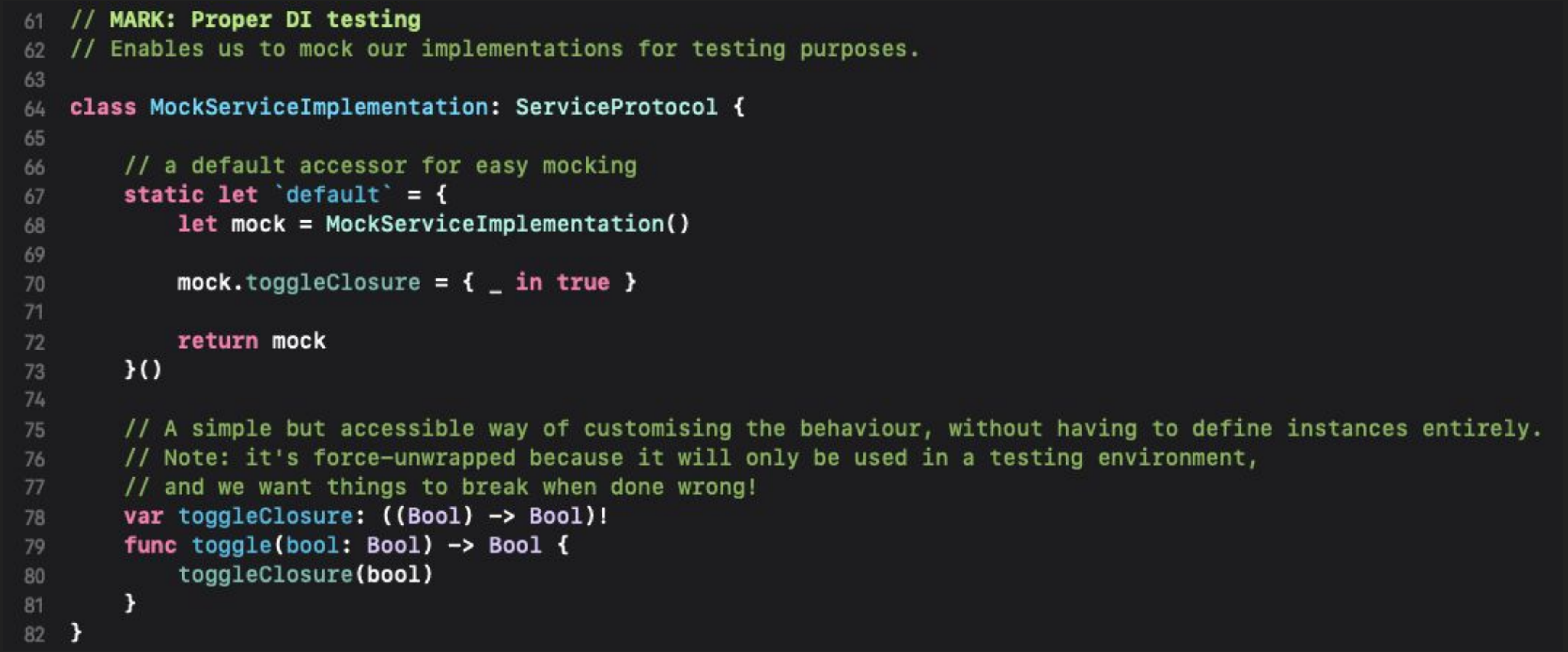
```swift
// MARK: Naive DI testing
// Enables us to stub our implementation for testing purposes 🙌

class NaiveStubServiceImplementation: ServiceProtocol {

    func toggle(bool: Bool) -> Bool {
        true
    }
}
```

Service Stubbing



```swift
// MARK: Naive DI testing
// Enables us to stub our implementation for testing purposes 🙌

class NaiveStubServiceImplementation: ServiceProtocol {

    func toggle(bool: Bool) -> Bool {
        true
    }
}

// Doesn't allow us to mock the inner workings without replicating the entirety of the implementation 😞

class NaiveStub2ServiceImplementation: ServiceProtocol {

    func toggle(bool: Bool) -> Bool {
        false
    }
}
```

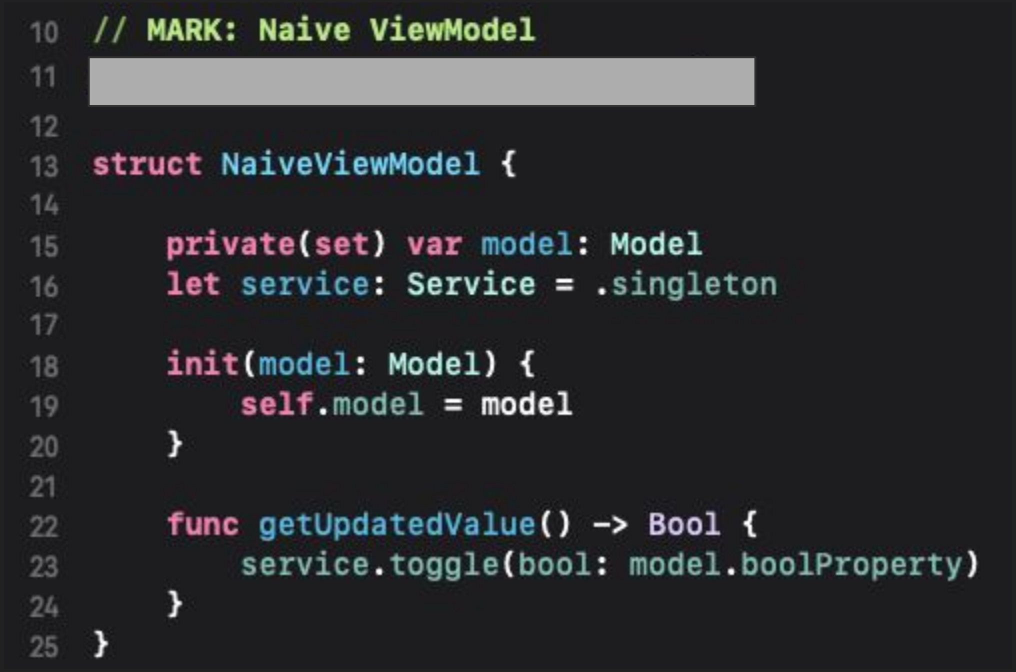
Service Mocking



```swift
// MARK: Proper DI testing
// Enables us to mock our implementations for testing purposes.

class MockServiceImplementation: ServiceProtocol {

    // a default accessor for easy mocking
    static let `default` = {
        let mock = MockServiceImplementation()

        mock.toggleClosure = { _ in true }

        return mock
    }()

    // A simple but accessible way of customising the behaviour, without having to define instances entirely.
    // Note: it's force-unwrapped because it will only be used in a testing environment,
    // and we want things to break when done wrong!
    var toggleClosure: ((Bool) -> Bool)!
    func toggle(bool: Bool) -> Bool {
        toggleClosure(bool)
    }
}
```

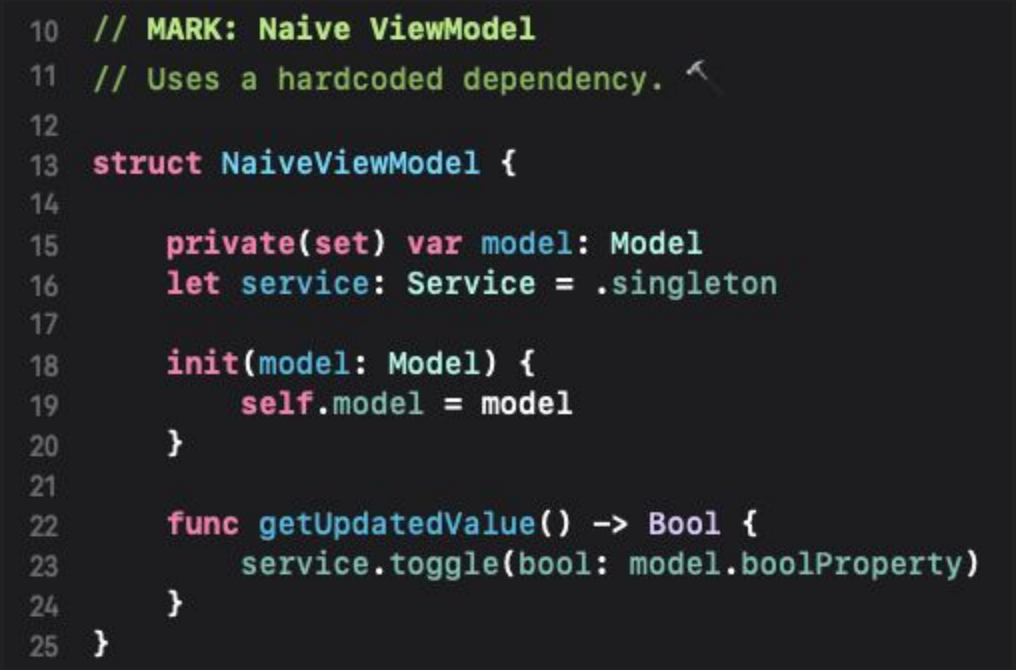
ViewModel DI

ViewModel



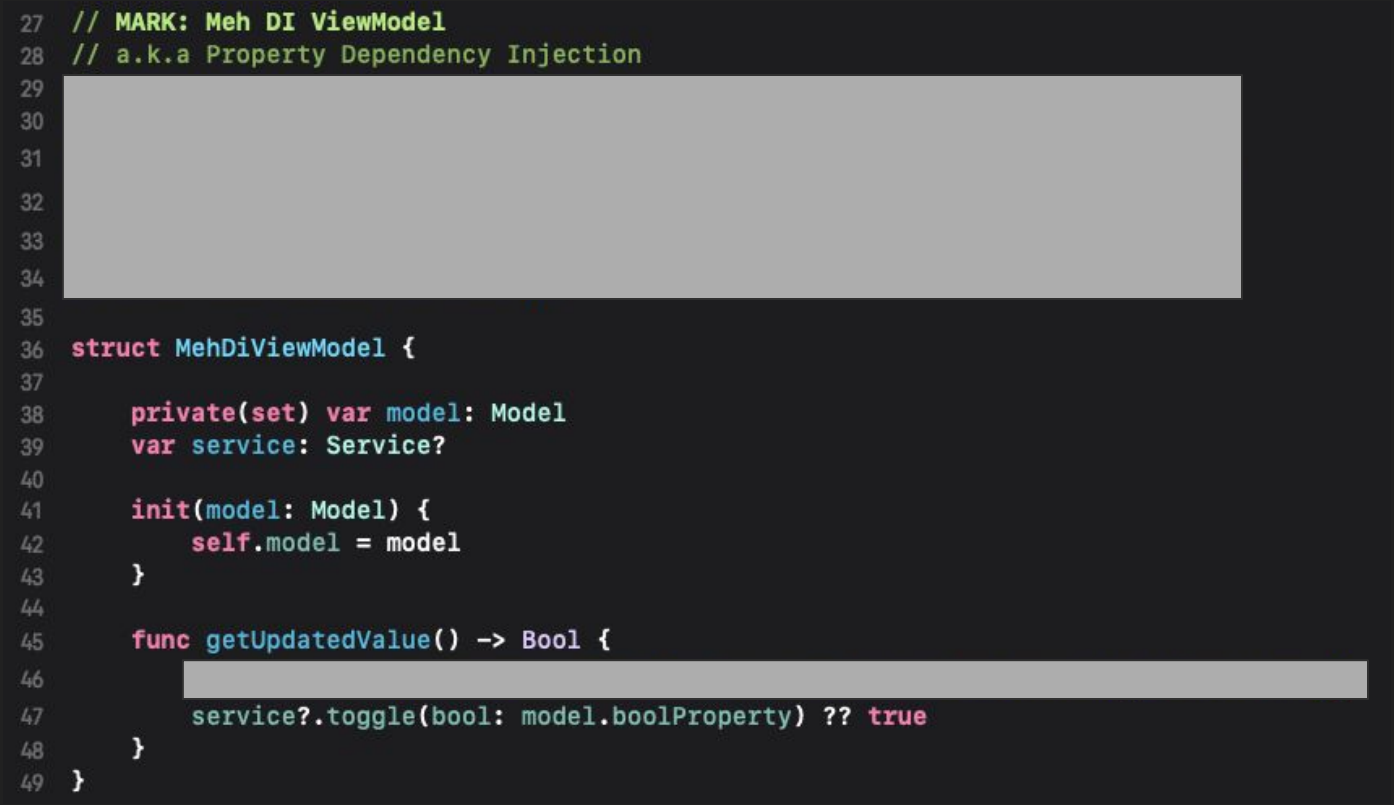
```swift
// MARK: Naive ViewModel
[redacted]

struct NaiveViewModel {

    private(set) var model: Model
    let service: Service = .singleton

    init(model: Model) {
        self.model = model
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel



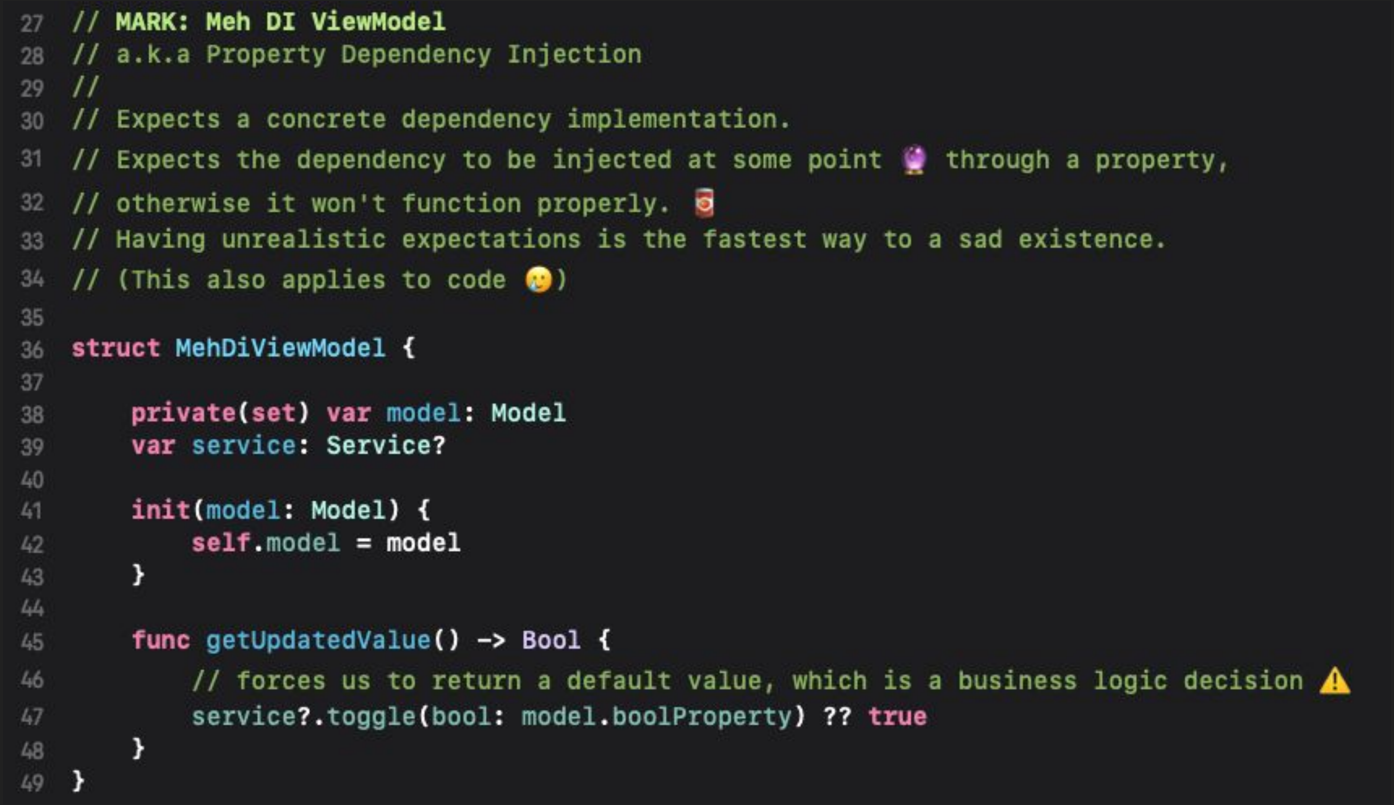
```swift
// MARK: Naive ViewModel
// Uses a hardcoded dependency.

struct NaiveViewModel {

    private(set) var model: Model
    let service: Service = .singleton

    init(model: Model) {
        self.model = model
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel



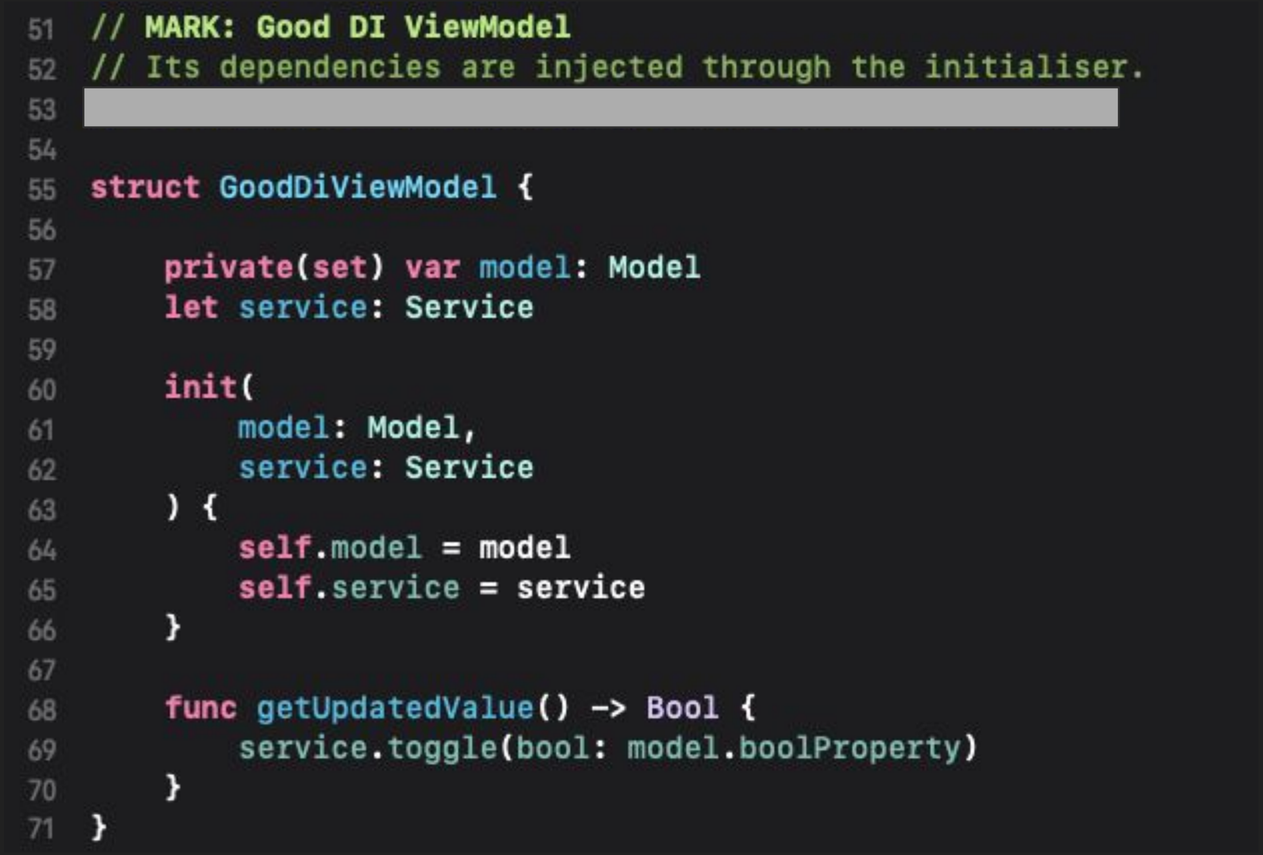
```swift
// MARK: Meh DI ViewModel
// a.k.a Property Dependency Injection
[redacted]

struct MehDiViewModel {

    private(set) var model: Model
    var service: Service?

    init(model: Model) {
        self.model = model
    }

    func getUpdatedValue() -> Bool {
[redacted]
        service?.toggle(bool: model.boolProperty) ?? true
    }
}
```

ViewModel



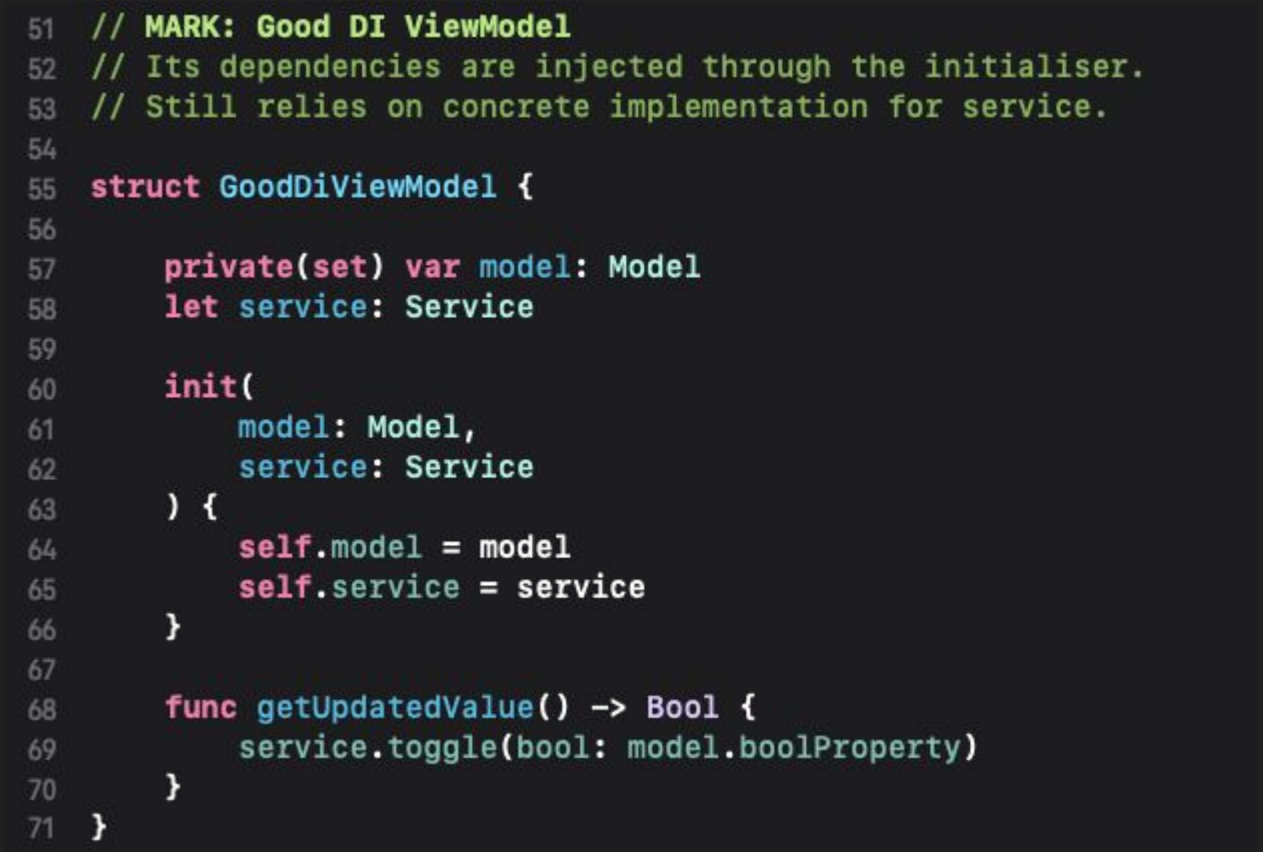
```swift
// MARK: Meh DI ViewModel
// a.k.a Property Dependency Injection
//
// Expects a concrete dependency implementation.
// Expects the dependency to be injected at some point 🔮 through a property,
// otherwise it won't function properly. 🥫
// Having unrealistic expectations is the fastest way to a sad existence.
// (This also applies to code 😢)

struct MehDiViewModel {

    private(set) var model: Model
    var service: Service?

    init(model: Model) {
        self.model = model
    }

    func getUpdatedValue() -> Bool {
        // forces us to return a default value, which is a business logic decision ⚠️
        service?.toggle(bool: model.boolProperty) ?? true
    }
}
```

ViewModel



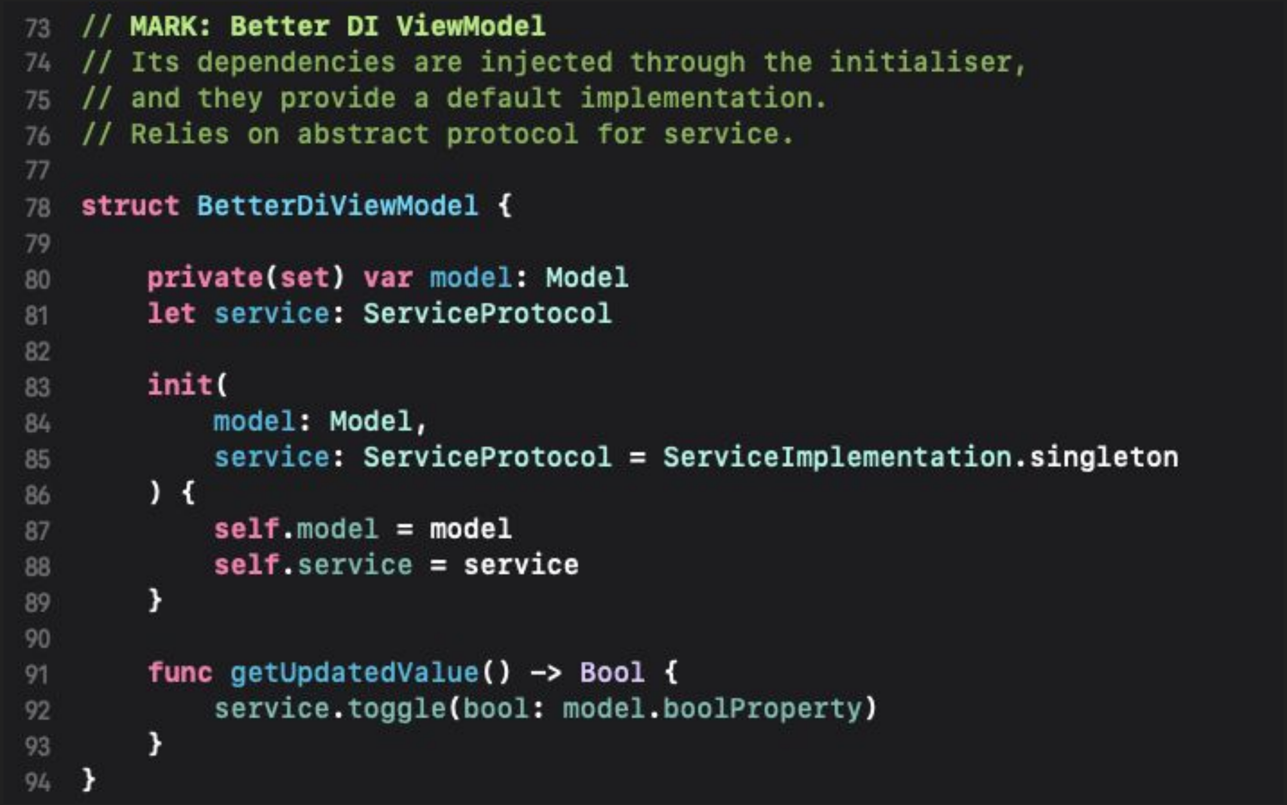
```swift
// MARK: Good DI ViewModel
// Its dependencies are injected through the initialiser.


struct GoodDiViewModel {

    private(set) var model: Model
    let service: Service

    init(
        model: Model,
        service: Service
    ) {
        self.model = model
        self.service = service
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel



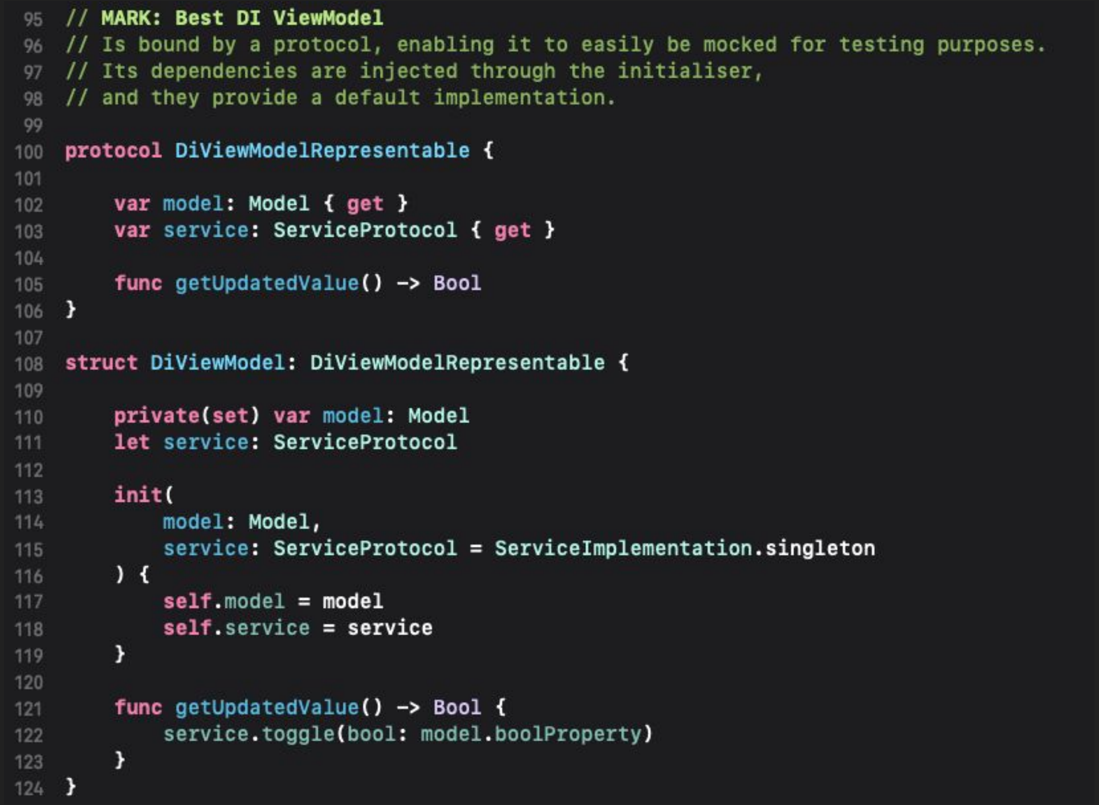
```swift
// MARK: Good DI ViewModel
// Its dependencies are injected through the initialiser.
// Still relies on concrete implementation for service.

struct GoodDiViewModel {

    private(set) var model: Model
    let service: Service

    init(
        model: Model,
        service: Service
    ) {
        self.model = model
        self.service = service
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel



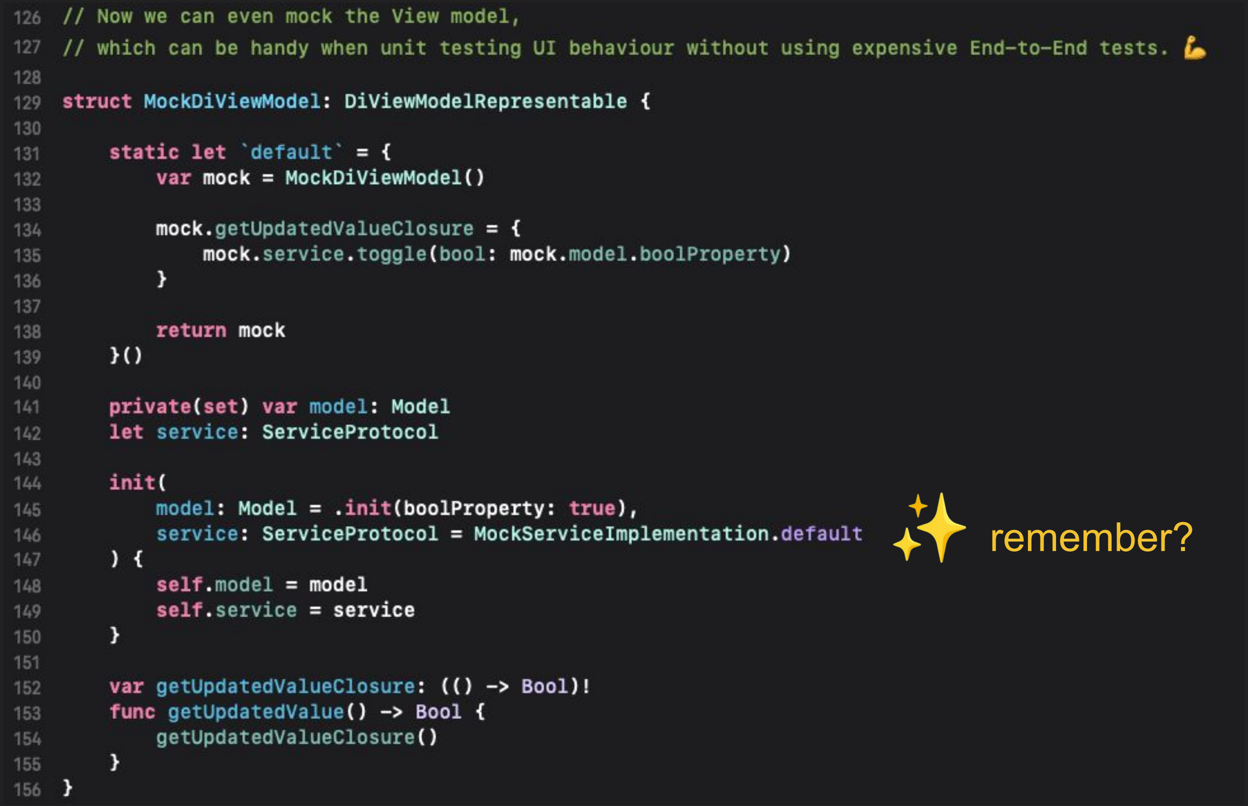
```swift
// MARK: Better DI ViewModel
// Its dependencies are injected through the initialiser,
// and they provide a default implementation.
// Relies on abstract protocol for service.

struct BetterDiViewModel {

    private(set) var model: Model
    let service: ServiceProtocol

    init(
        model: Model,
        service: ServiceProtocol = ServiceImplementation.singleton
    ) {
        self.model = model
        self.service = service
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel



```swift
// MARK: Best DI ViewModel
// Is bound by a protocol, enabling it to easily be mocked for testing purposes.
// Its dependencies are injected through the initialiser,
// and they provide a default implementation.

protocol DiViewModelRepresentable {

    var model: Model { get }
    var service: ServiceProtocol { get }

    func getUpdatedValue() -> Bool
}

struct DiViewModel: DiViewModelRepresentable {

    private(set) var model: Model
    let service: ServiceProtocol

    init(
        model: Model,
        service: ServiceProtocol = ServiceImplementation.singleton
    ) {
        self.model = model
        self.service = service
    }

    func getUpdatedValue() -> Bool {
        service.toggle(bool: model.boolProperty)
    }
}
```

ViewModel Mocking



```swift
// Now we can even mock the View model,
// which can be handy when unit testing UI behaviour without using expensive End-to-End tests. 💪

struct MockDiViewModel: DiViewModelRepresentable {

    static let `default` = {
        var mock = MockDiViewModel()

        mock.getUpdatedValueClosure = {
            mock.service.toggle(bool: mock.model.boolProperty)
        }

        return mock
    }()

    private(set) var model: Model
    let service: ServiceProtocol

    init(
        model: Model = .init(boolProperty: true),
        service: ServiceProtocol = MockServiceImplementation.default
    ) {
        self.model = model
        self.service = service
    }

    var getUpdatedValueClosure: (() -> Bool)!
    func getUpdatedValue() -> Bool {
        getUpdatedValueClosure()
    }
}
```

✨ remember?

What are the pitfalls of our DI implementation?

What are the pitfalls of our DI implementation?

What’s the lifecycle of a `static` property?

What are the pitfalls of our DI implementation?

What's the lifecycle of a `static` property?

https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264 :

> Stored type properties are lazily initialized on their first access. They're guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they don't need to be marked with the lazy modifier.

What are the pitfalls of our DI implementation?

What’s the lifecycle of a `static` property?

https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264 :

> Stored type properties are lazily initialized on their first access. They’re guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they don’t need to be marked with the lazy modifier.

Why does this matter?

What are the pitfalls of our DI implementation?

What's the lifecycle of a `static` property?

https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264 :

> Stored type properties are lazily initialized on their first access. They're guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they don't need to be marked with the lazy modifier.

Why does this matter?

They never get cleaned up after being accessed the first time,
until you terminate the app! So if we have a bunch of huge services… 💥

Service Locators

Service Locators - Pseudo-Code

- Hold all of the services that should be deallocated somewhere (stored in a collection, e.g. Dictionary key-value)
- Have something to manage and search for services in that collection

ServiceLocator.swift

```
- var services = [ServiceName : Any]()
- register(Service)
- get(Service)
- unregister(Service)
```

Service Locator Code

How many lines do you think it will take? 😱

Unit Tests 💪

Reading Materials

Wikipedia

- https://en.wikipedia.org/wiki/Dependency_inversion_principle
- https://en.wikipedia.org/wiki/Dependency_injection
- https://en.wikipedia.org/wiki/Service_locator_pattern

Stored properties - https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID264

Service Locators

- https://github.com/Mindera/Alicerce/blob/408a3015dc578f2598c14645b942f04c9042d7ce/Sources/Utils/ServiceLocator.swift
- https://github.com/Mindera/Alicerce/blob/408a3015dc578f2598c14645b942f04c9042d7ce/Tests/AlicerceTests/Utils/ServiceLocatorTests.swift
- https://quickbirdstudios.com/blog/swift-dependency-injection-service-locators/
- https://stevenpcurtis.medium.com/the-service-locator-pattern-in-swift-5db2c770bcc
- https://www.oracle.com/java/technologies/service-locator.html
- https://www.baeldung.com/java-service-locator-pattern

Questions?

Thank you for coming to my TED talk