디자인 패턴

Facade, Proxy, Decorator, Flyweight

Effie, Paul

목차

- 용어와 유의사항 짚고 가기
- Facade
- Proxy
- Decorator
- Flyweight

짚고 가기
유의사항들 ...

- 넘 빠를 수 있어요.
- 예시 코드들은 활용 보다는 패턴에 대한 이해를 도모하는 데 목적이 있어요.
- 중간 중간 질문 주셔도 돼요. 중간 질문이 더 좋을 수도 있어요.
  - ex. 자꾸 나오는 클라이언트가 뭘 말하는 거죠?
  - 대신 질문이 다 답변이 될 거라는 보장은 없습니다 ;)
  - 그리고 제가 답변을 잠깐 미룰 수도 있어요.

짚고 가기
엄청 자주 나올 용어들

- 서비스: API, 라이브러리, 프레임워크 등 기능을 제공하는 party
- 클라이언트: 서비스를 쓰는 party
- 컴파일 타임
  - 타입으로, 구현으로 고정된다.
- 런타임
  - 상태의 영향을 받는다.

Facade

문제

```swift
import CoreLocation

func 현재위치의_주소찾기() {
  if let lastLocation = CLLocationManager().location {
    CLGeocoder().reverseGeocodeLocation(
      lastLocation,
      completionHandler: { (placemarks, error) in
        if error == nil {
          guard
            let firstLocation = placemarks?[0],
            let name = firstLocation.name
          else {
            print("현재 위치를 찾을 수 없습니다.")
            return
          }
          print("현재 위치는 \(name)입니다.")
        } else {
          print("현재 위치를 찾을 수 없습니다.")
        }
      }
    )
  } else {
    print("현재 위치를 찾을 수 없습니다.")
  }
}
```

```swift
import CoreLocation

func 현재주소_좌표찾기() {
  let currentAddressString = "현재 주소"

CLGeocoder().geocodeAddressString(currentAddressString)
{ (placemarks, error) in
    if error == nil {
      if let placemark = placemarks?[0],
          let location = placemark.location {
        let coord = (location.coordinate.latitude,
                     location.coordinate.longitude)
        print(coord)
        return
      }
    }
    print("찾을 수 없습니다")
  }
}
```

문제

- 서비스 인터페이스를 여기 저기서 호출하다 보면 코드가 너무 복잡해진다.
- 유지 보수도 어렵다.
- 모듈, 타입에 대한 의존성이 덕지덕지..

정의

- 라이브러리에 대한, 프레임워크에 대해...
- 또는 다른 클래스들의 복잡한 집합에 대해...
- 단순화된 인터페이스를 제공하는 구조적 디자인 패턴

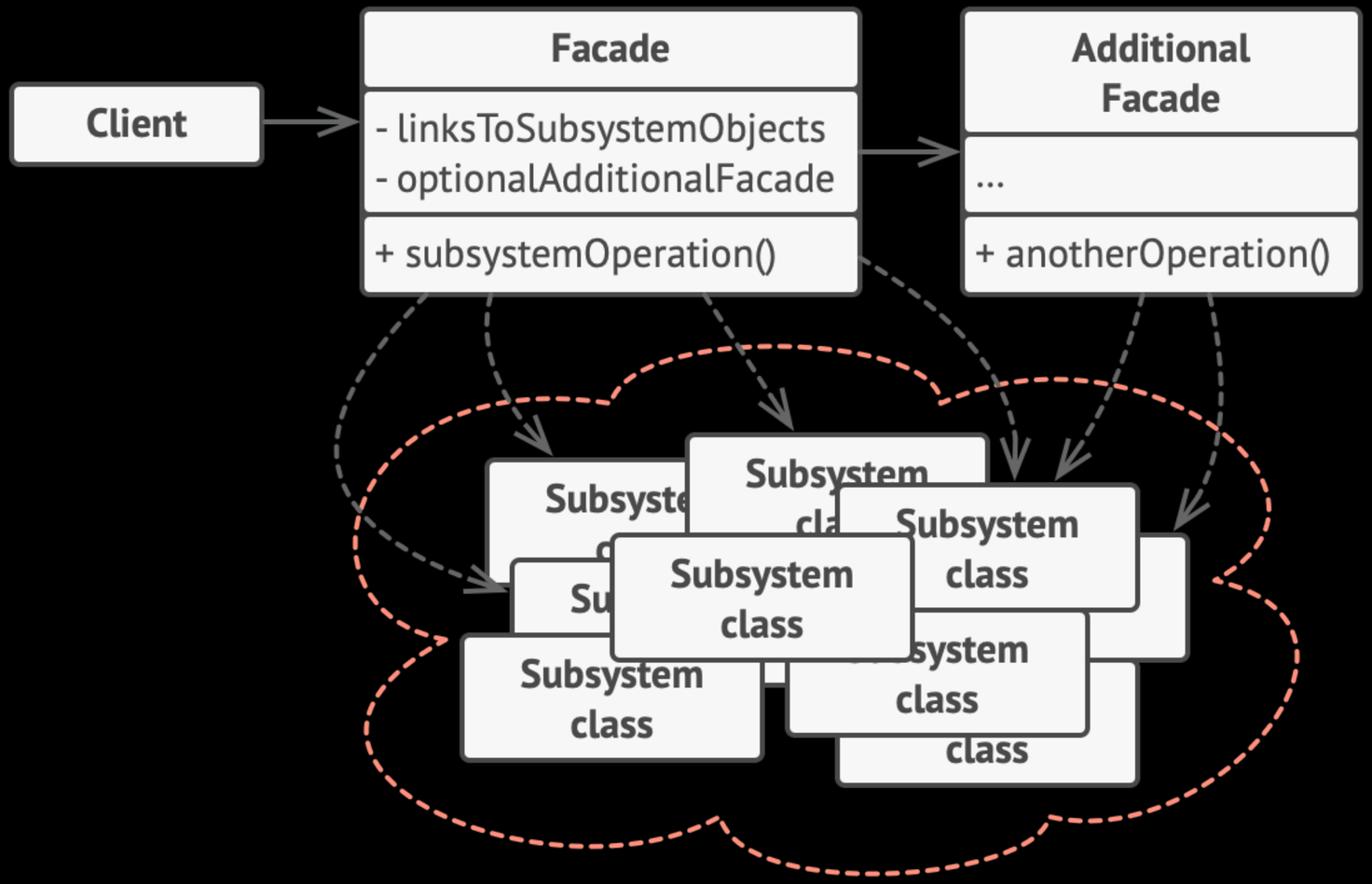
정의

facade ★

1. 명사 (건물의) 정면[앞면]
2. 명사 (실제와는 다른) 표면, 허울

- 정면, 겉
- 복잡한 시스템으로 진입하는 관문

구성 요소


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

다시.. 문제의 코드

```swift
import CoreLocation

func 현재위치의_주소찾기() {
  if let lastLocation = CLLocationManager().location {
    CLGeocoder().reverseGeocodeLocation(
      lastLocation,
      completionHandler: { (placemarks, error) in
        if error == nil {
          guard
            let firstLocation = placemarks?[0],
            let name = firstLocation.name
          else {
            print("현재 위치를 찾을 수 없습니다.")
            return
          }
          print("현재 위치는 \(name)입니다.")
        } else {
          print("현재 위치를 찾을 수 없습니다.")
        }
      }
    )
  } else {
    print("현재 위치를 찾을 수 없습니다.")
  }
}
```

```swift
import CoreLocation

func 현재주소_좌표찾기() {
  let currentAddressString = "현재 주소"

CLGeocoder().geocodeAddressString(currentAddressString)
{ (placemarks, error) in
    if error == nil {
      if let placemark = placemarks?[0],
          let location = placemark.location {
        let coord = (location.coordinate.latitude,
                     location.coordinate.longitude)
        print(coord)
        return
      }
    }
    print("찾을 수 없습니다")
  }
}
```

다시.. 문제의 코드

```swift
import CoreLocation

func 현재위치의_주소찾기() {
  if let lastLocation = CLLocationManager().location {
    CLGeocoder().reverseGeocodeLocation(
      lastLocation,
      completionHandler: { (placemarks, error) in
        if error == nil {
          guard
            let firstLocation = placemarks?[0]
            let name = firstLocation.name
          else {
            print("현재 위치를 찾을 수 없습니다.")
            return
          }
          print("현재 위치는 \(name)입니다.")
        } else {
          print("현재 위치를 찾을 수 없습니다.")
        }
      }
    )
  } else {
    print("현재 위치를 찾을 수 없습니다.")
  }
}
```

Manager...
마렵다

LocationManager...

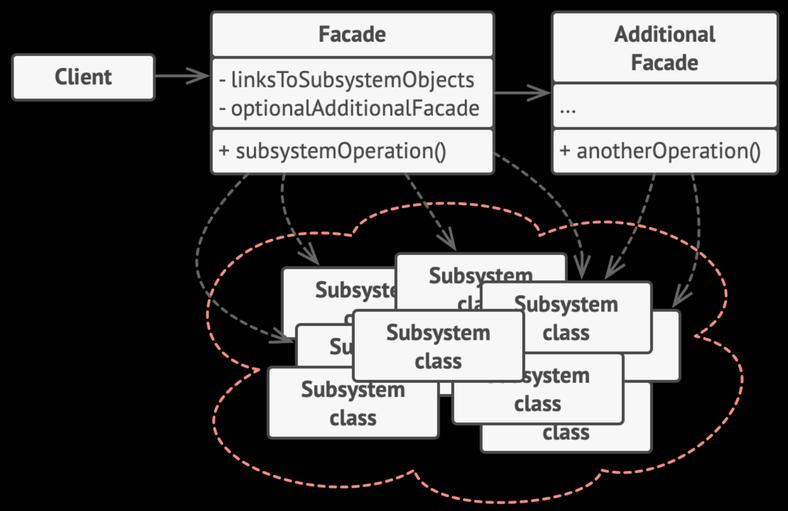
```swift
import CoreLocation

func 현재주소_좌표찾기() {
  let currentAddressString = "현재 주소"
  CLGeocoder().geocodeAddressString(currentAddressString)
  { (placemarks, error) in
    if error == nil {
      if let placemark = placemarks?[0],
         let location = placemark.location {
        let coord = (location.coordinate.latitude,
                     location.coordinate.longitude)
        print(coord)
        return
      }
    }
    print("찾을 수 없습니다")
  }
}
```

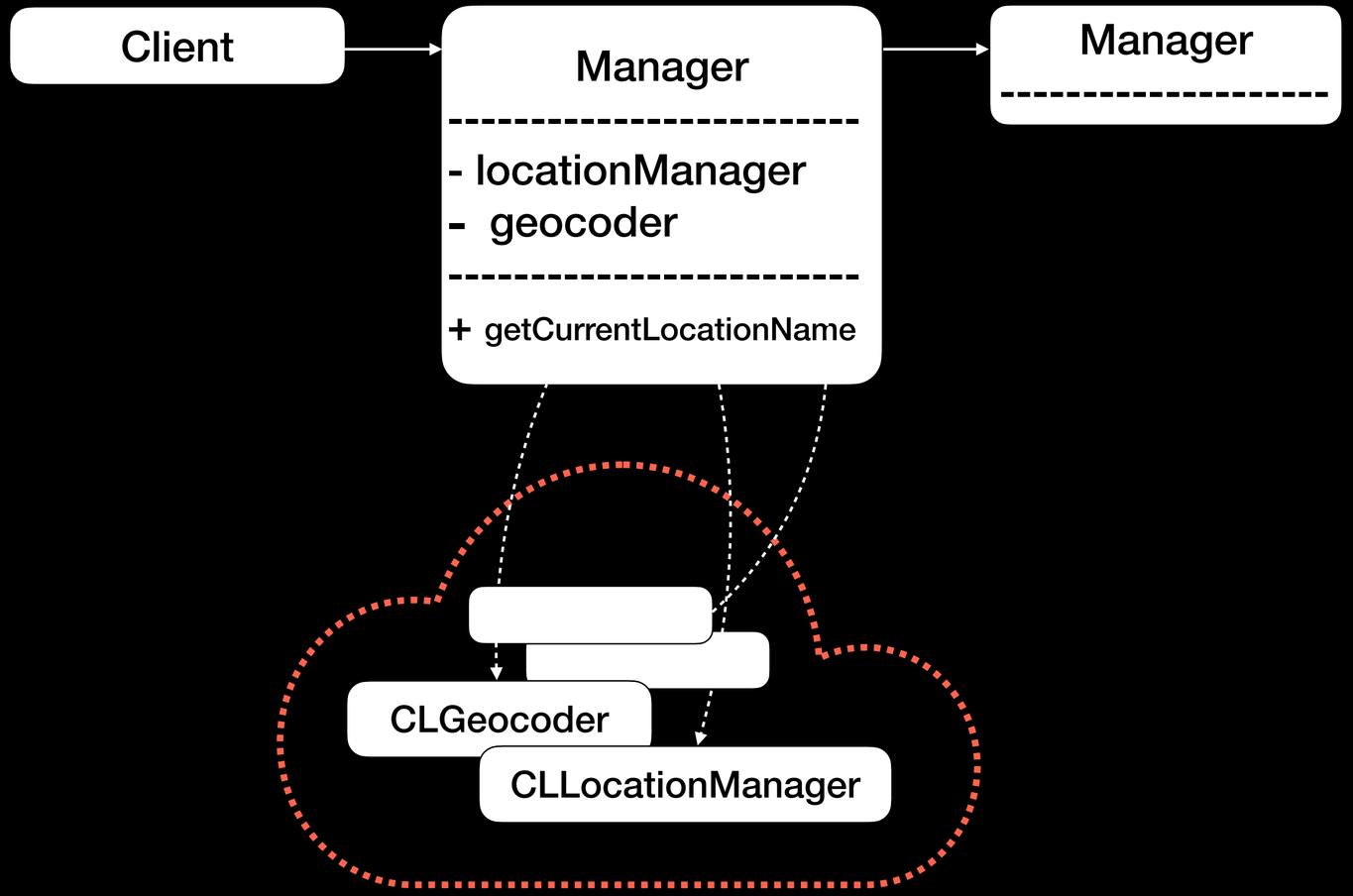
문제의 해결

```swift
import CoreLocation

func 현재위치의_주소찾기() {
  if let lastLocation = CLLocationManager().location {
    CLGeocoder().reverseGeocodeLocation(
      lastLocation,
      completionHandler: { (placemarks, error) in
        if error == nil {
          guard
            let firstLocation = placemarks?[0],
            let name = firstLocation.name
          else {
            print("현재 위치를 찾을 수 없습니다.")
            return
          }
          print("현재 위치는 \(name)입니다.")
        } else {
          print("현재 위치를 찾을 수 없습니다.")
        }
      }
    )
  } else {
    print("현재 위치를 찾을 수 없습니다.")
  }
}
```

```swift
import CoreLocation

final class CoreLocationFacade {
  private let locationManager = CLLocationManager()

  private let geocoder: CLGeocoder

  init(geocoder: CLGeocoder = CLGeocoder()) {
    self.geocoder = geocoder
  }

  func getCurrentLocationName(
    completionHandler: @escaping (String?) -> Void
  ) {
    if let lastLocation = self.locationManager.location {
      self.geocoder.reverseGeocodeLocation(
        lastLocation,
        completionHandler: { (placemarks, error) in
          if error == nil {
            let firstLocation = placemarks?[0]
            completionHandler(firstLocation?.name)
          } else {
            completionHandler(nil)
          }
        }
      )
    } else {
      completionHandler(nil)
    }
  }
  ...
```

구성 요소


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class


Client
Manager
- locationManager
- geocoder
+ getCurrentLocationName
Manager
CLGeocoder
CLLocationManager

장단점과 주의 사항

- 복잡한 하위 시스템에서 코드를 별도로 분리할 수 있습니다.
- 앱의 모든 클래스에 결합된 **전지전능한 객체** (God Object)가 될 수 있습니다.
  - SRP...!
- 기존 하위시스템이 이미 제공하고 있는 것보다 더 간단한 인터페이스를 제공하는 것이 가능한지 확인하세요. 이 인터페이스가 클라이언트 코드를 하위시스템의 여러 클래스로부터 독립시킨다면 제대로 하고 계신 겁니다.

정리

- 복잡한 시스템으로 진입하는 관문
- Complex? Simplify it.

Proxy

문제

API... 그냥은 못 쓰겠다

- 이 API 그냥 사용해도 되는 거야? 상태에 따른 분기가 필요한데..
- 이 서비스 지금 쓰지도 않는데 초기화한다고? 메모리 엄청 차지 할텐데... ex. DB
- API 호출할 때마다 기록 좀 남기고 싶다...
- 호출한 결과 캐싱 마렵다..
- 등등

정의

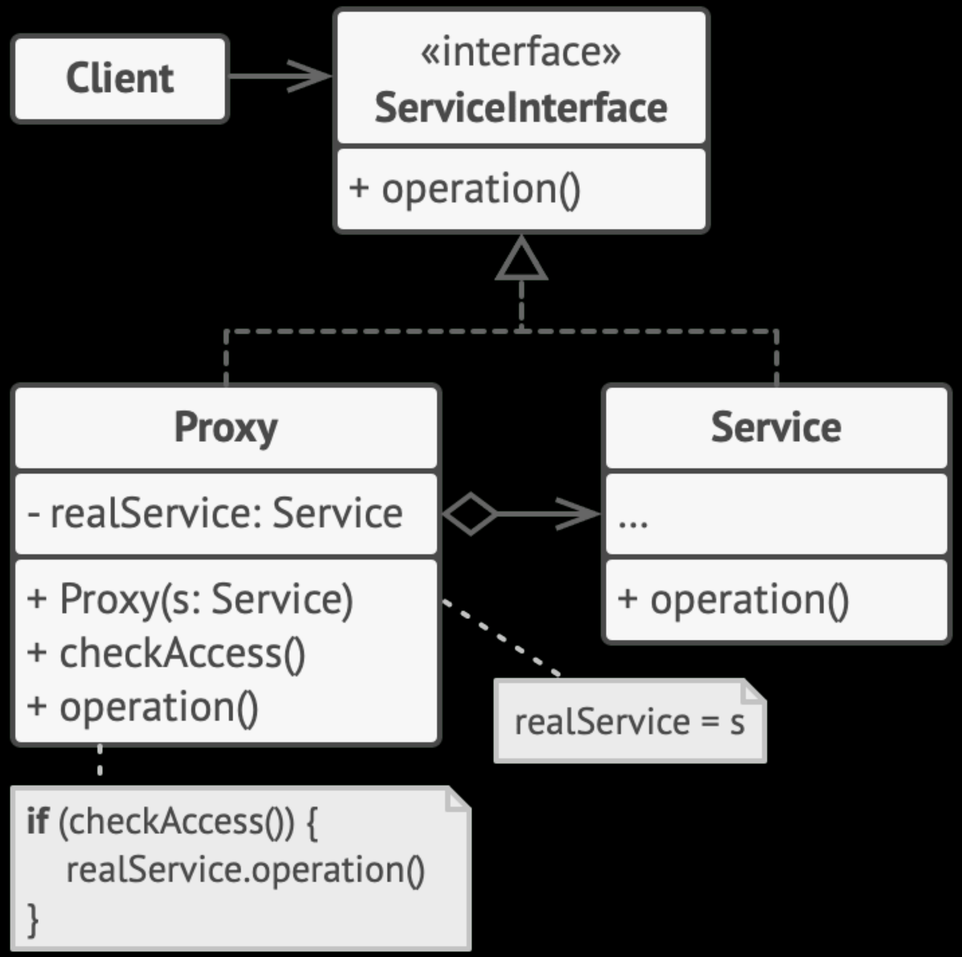
- 프록시 객체는 실제 서비스와 같은 인터페이스를 사용해서
- 클라이언트가 서비스 객체를 알지 못한 채 (직접 접근하지 않고)
- 서비스 객체의 기능을 사용할 수 있는 객체!
- 실제 서비스 객체의 대타 역할을 한다. (타입이 같기 때문에)

정의

proxy

1. 명사 대리[위임](권)
2. 명사 대리인
3. 명사 (측정·계산하려는 다른 것을 대표하도록 이용하는) 대용물

구성 요소


Client
«interface»
ServiceInterface
+ operation()
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
Service
...
+ operation()
realService = s
if (checkAccess()) {
realService.operation()
}

문제의 해결

```
final class GithubClass {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/commits"
    // ...
  }

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/branches"
    // ...
  }

  // ...
}
```

문제의 해결

```swift
final class GithubClass {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/commits"
    // ...
  }

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/branches"
    // ...
  }

  // ...
}
```

```swift
// 서비스와 같은 기능을 제공하는 인터페이스를 만듭니다.
protocol GithubProtocol {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )
}
```

문제의 해결

```swift
final class GithubClass {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/commits"
    // ...
  }

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  ) {
    let urlString = "https://api.github.com/repos/\
(owner)/\(repo)/branches"
    // ...
  }

  // ...
}
```

```swift
// 서비스와 같은 기능을 제공하는 인터페이스를 만듭니다.
protocol GithubProtocol {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )
}
```

```swift
// 서비스는 이미 해당 인터페이스를 제공하고 있습니다!
extension GithubClass: GithubProtocol { }
```

문제의 해결

```swift
// 서비스와 같은 기능을 제공하는 인터페이스를 만듭니다.
protocol GithubProtocol {
  func commit(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )

  func branch(
    owner: Owner,
    repo: Repository,
    auth: Authorization
  )
}
```

```swift
// 서비스는 이미 해당 인터페이스를 제공하고 있습니다!
extension GithubClass: GithubProtocol { }
```

```swift
final class LogGithubProxy: GithubProtocol {
  private let service: GithubProtocol

  private let logger: Logger

  init(service: GithubProtocol, logger: Logger) {
    self.service = service
    self.logger = logger
  }

  func commit(owner: Owner, repo: Repository, auth:
Authorization) {
    self.logger.log("will commit @\(owner)'s repo \(repo)")

    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.logger.log("will commit @\(owner)'s repo \(repo)")
  }

  func branch(owner: Owner, repo: Repository, auth:
Authorization) {
    self.logger.log("will make branch @\(owner)'s repo \
(repo)")

    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.logger.log("will make branch @\(owner)'s repo \
(repo)")
  }
}
```

문제의 해결

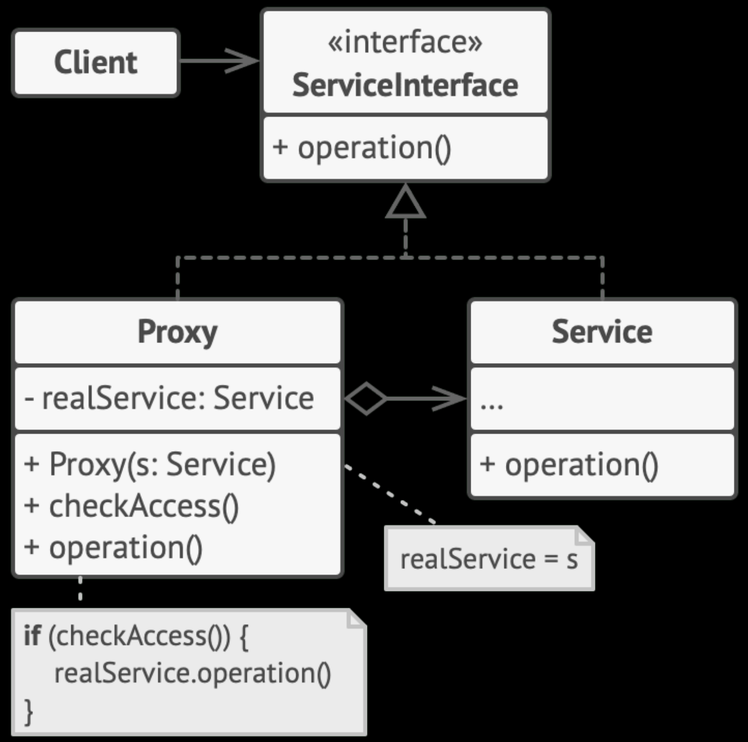
```
final class LogGithubProxy: GithubProtocol {
  private let service: GithubProtocol

  private let logger: Logger

  init(service: GithubProtocol, logger: Logger) {
    self.service = service
    self.logger = logger
  }

  func commit(owner: Owner, repo: Repository, auth:
Authorization) {
    self.logger.log("will commit @\(owner)'s repo \(repo)")

    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.logger.log("will commit @\(owner)'s repo \(repo)")
  }

  func branch(owner: Owner, repo: Repository, auth:
Authorization) {
    self.logger.log("will make branch @\(owner)'s repo \
(repo)")

    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.logger.log("will make branch @\(owner)'s repo \
(repo)")
  }
}
```

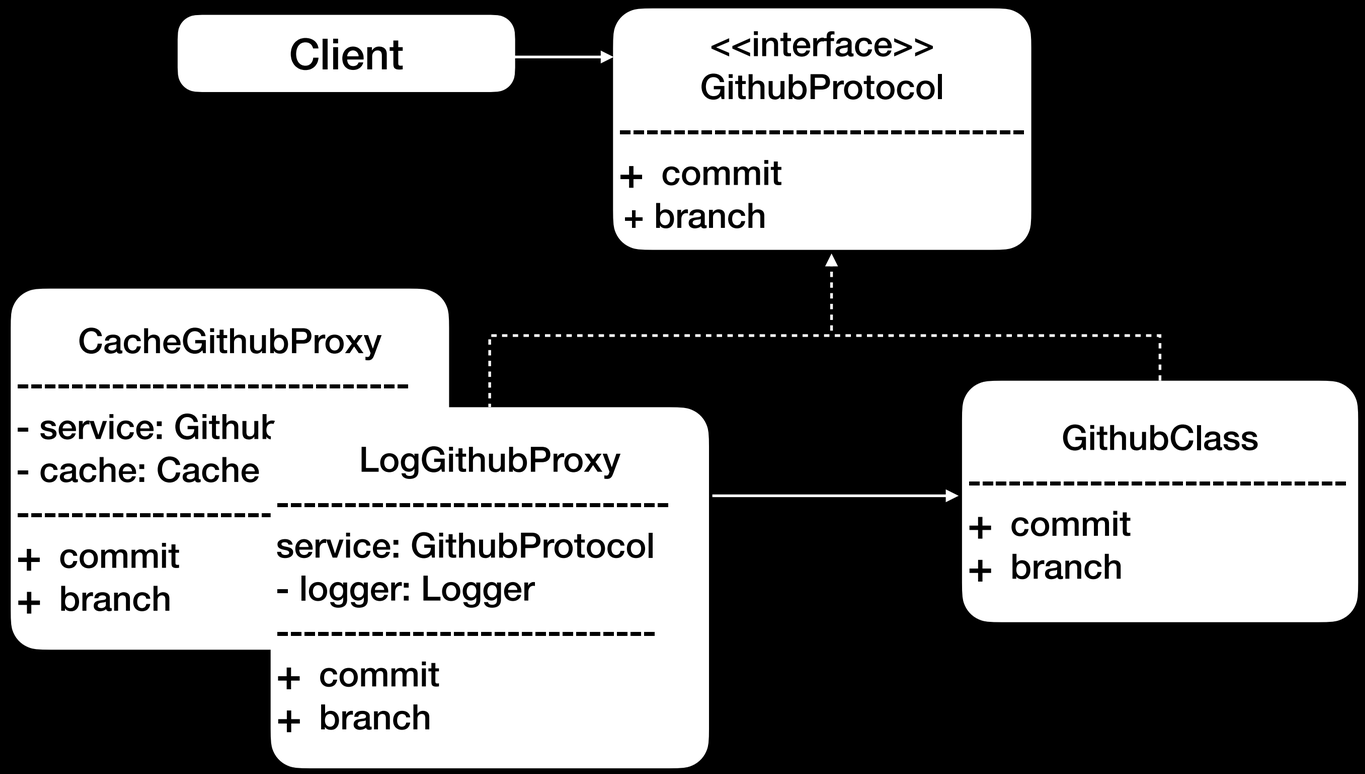
```
final class CacheGithubProxy: GithubProtocol {
  private let service: GithubProtocol = GithubClass()

  private let cache: CacheManager = CacheManager()

  func commit(owner: Owner, repo: Repository, auth:
Authorization) {
    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.cache.cache()
  }

  func branch(owner: Owner, repo: Repository, auth:
Authorization) {
    self.service.commit(owner: owner, repo: repo, auth:
auth)

    self.cache.cache()
  }
}
```

문제의 해결

```
var githubService: GithubProtocol = GithubClass()
  githubService.commit(owner: "hyeffie", repo:
"BoxOffice", auth: "")

  githubService = LogGithubProxy(service:
githubService, logger: .init())
  githubService.commit(owner: "hyeffie", repo:
"BoxOffice", auth: "")

  githubService = CacheGithubProxy()
  githubService.commit(owner: "hyeffie", repo:
"BoxOffice", auth: "")
```

구성 요소


Client
«interface»
ServiceInterface
+ operation()
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
Service
...
+ operation()
realService = s
if (checkAccess()) {
realService.operation()
}


Client
<<interface>>
GithubProtocol
+ commit
+ branch
CacheGithubProxy
- service: Github
- cache: Cache
+ commit
+ branch
LogGithubProxy
service: GithubProtocol
- logger: Logger
+ commit
+ branch
GithubClass
+ commit
+ branch

파사드 vs. 프록시

- 공통점
  - 클라이언트가 서비스에 직접 접근하지 않게 된다. (서비스 객체를 몰라도 된다)
- 차이점
  - 프록시는 서비스를 대체하기 위한 패턴이기 때문에 반드시 서비스와 공통 인터페이스를 만들어야 한다.
  - 파사드는 인터페이스가 같아야 할 필요는 없다. (애초에 복잡한 시스템을 단순화하는 것이 목적)

여러 가지 프록시

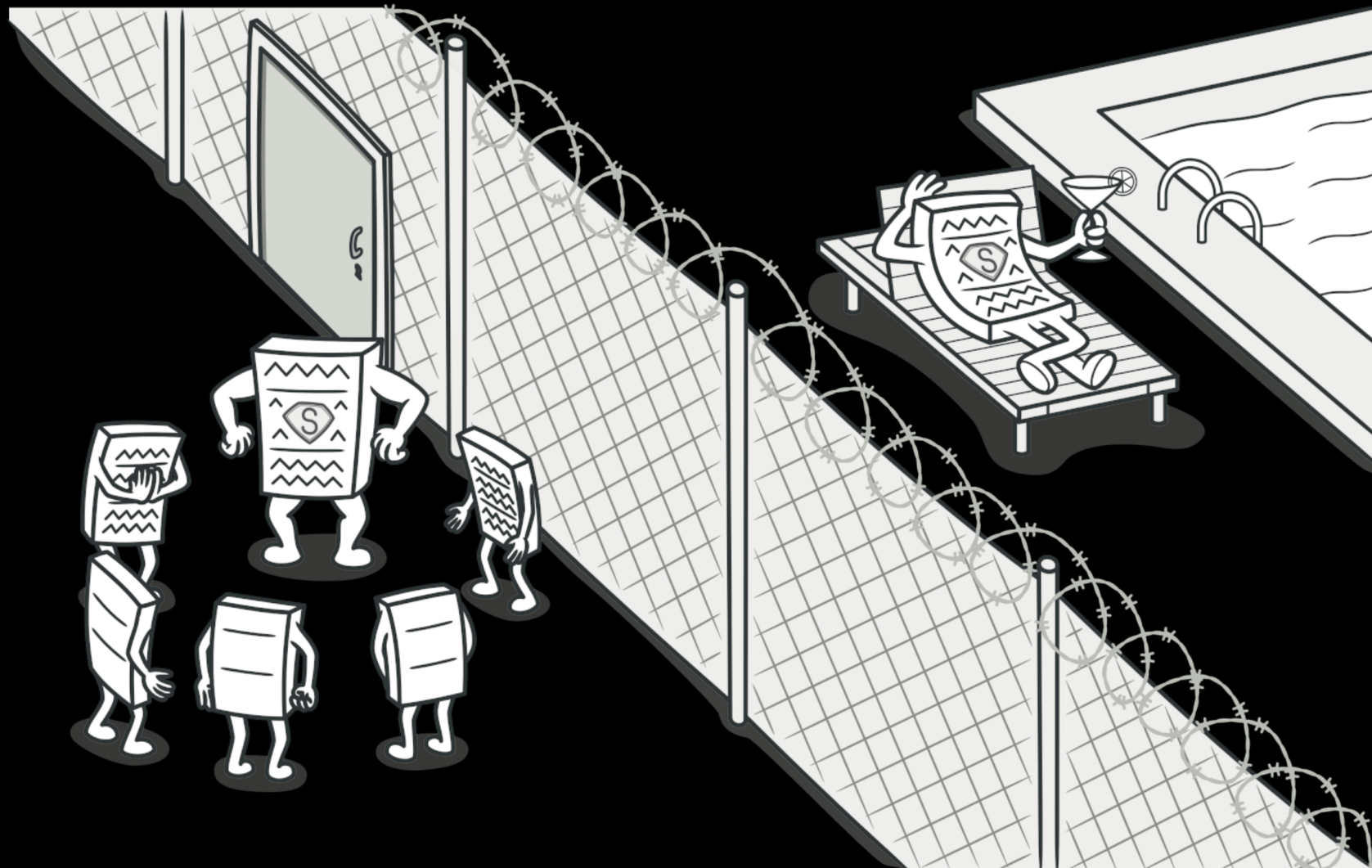
- 가상 프록시
  - 서비스 객체의 초기화가 무겁다면 서비스 객체 초기화를 지연하는 가상 프록시를 만들 수 있다.
- 보호 프록시
  - 서비스 객체의 사용을 보호하고자 할 때, 조건에 따라 요청 전달을 결정하는 보호 프록시를 만들 수 있다.
- 원격 프록시
  - 네트워크 (원격) 서비스를 사용할 때… 원격 프록시를 만들어 로컬처럼 보이게 할 수 있다.
- 로깅 프록시
  - 서비스를 요청하기 전에 로깅하고자 할 때 로깅 프록시
- 캐싱 프록시
  - 같은 요청에 대해 결과를 캐싱하고자 할 때… 캐싱 프록시
- 스마트 참조
  - 서비스 객체나 서비스 결과에 대한 참조를 얻은 클라이언트를 추적해서 불필요한 자원 낭비를 막고자 할 때…! 스마트 참조 (Smart reference)

장단점과 주의 사항

- 클라이언트를 서비스와 분리할 수 있다
  - 클라이언트들이 알지 못하는 상태에서 서비스 객체를 제어할 수 있습니다.
  - 클라이언트들이 신경 쓰지 않을 때 서비스 객체의 수명 주기를 관리할 수 있습니다.
  - 프록시는 서비스 객체가 준비되지 않았거나 사용할 수 없는 경우에도 작동합니다.
- OCP: 개방/폐쇄 원칙
  - 서비스나 클라이언트들을 변경하지 않고도 새 프록시들을 도입할 수 있습니다.
- 구현 비용
  - 새로운 클래스들을 많이 도입해야 하므로 코드가 복잡해질 수 있습니다.
- 응답 지연
  - 서비스의 응답이 늦어질 수 있습니다.

정리

- 서비스 대타



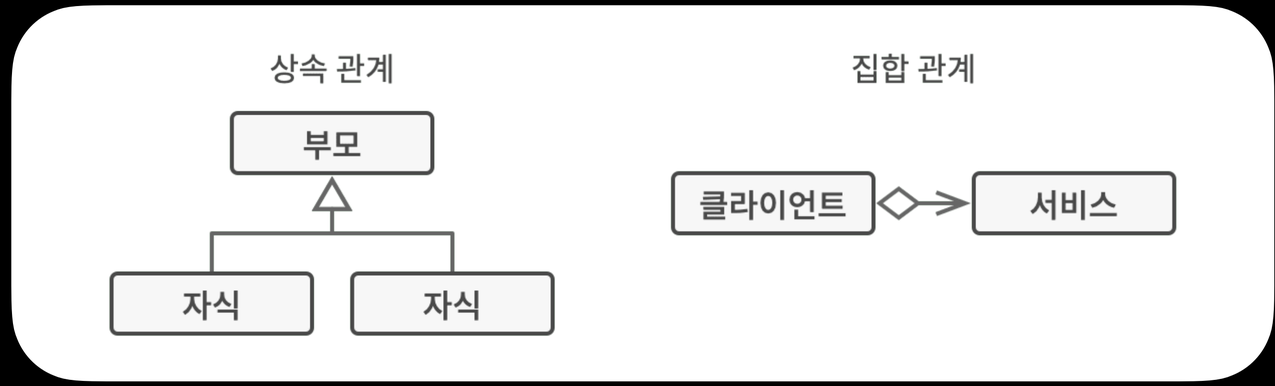
Decorator

문제

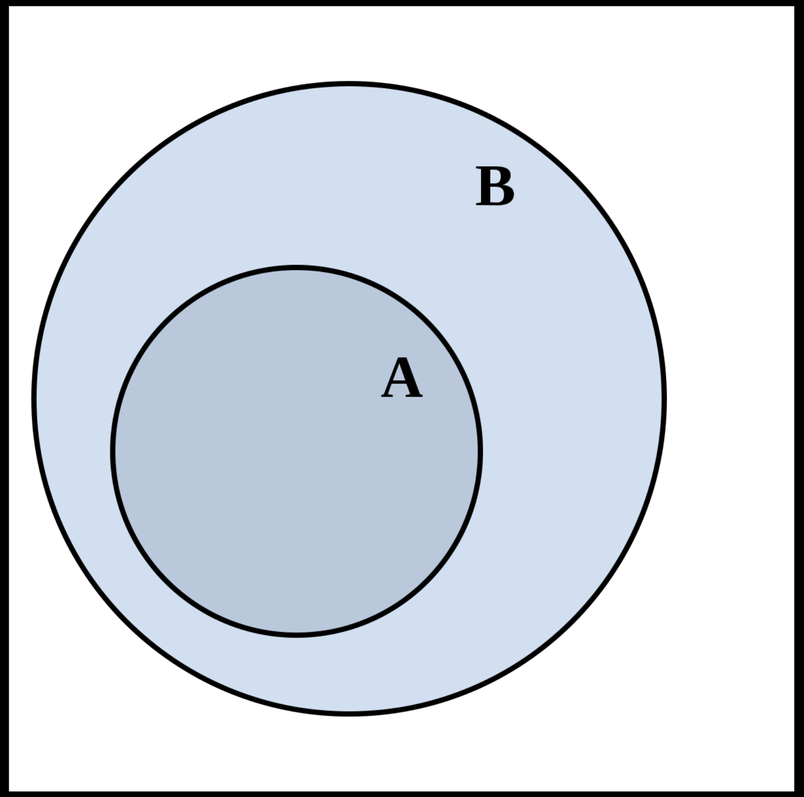
- 이미 있는 클래스에 기능을 추가해서 사용하고 싶다 > 서브클래싱 !
- 그런데 여러 가지 기능을 한 번에 사용하고 싶다면 ? > 경우의 수만큼 클래스 선언?
- 상속이라 합성도 어렵군..
- 여러 모로 구현 비용이 만만치 않다...
- 나는 런타임에 상태에 따라서 필요한 만큼만 쓰고 싶었을 뿐인데...
- (기타 서브 클래싱(상속)의 여러 문제점들로 원하는 구조 구현이 어색하다...!)

정의

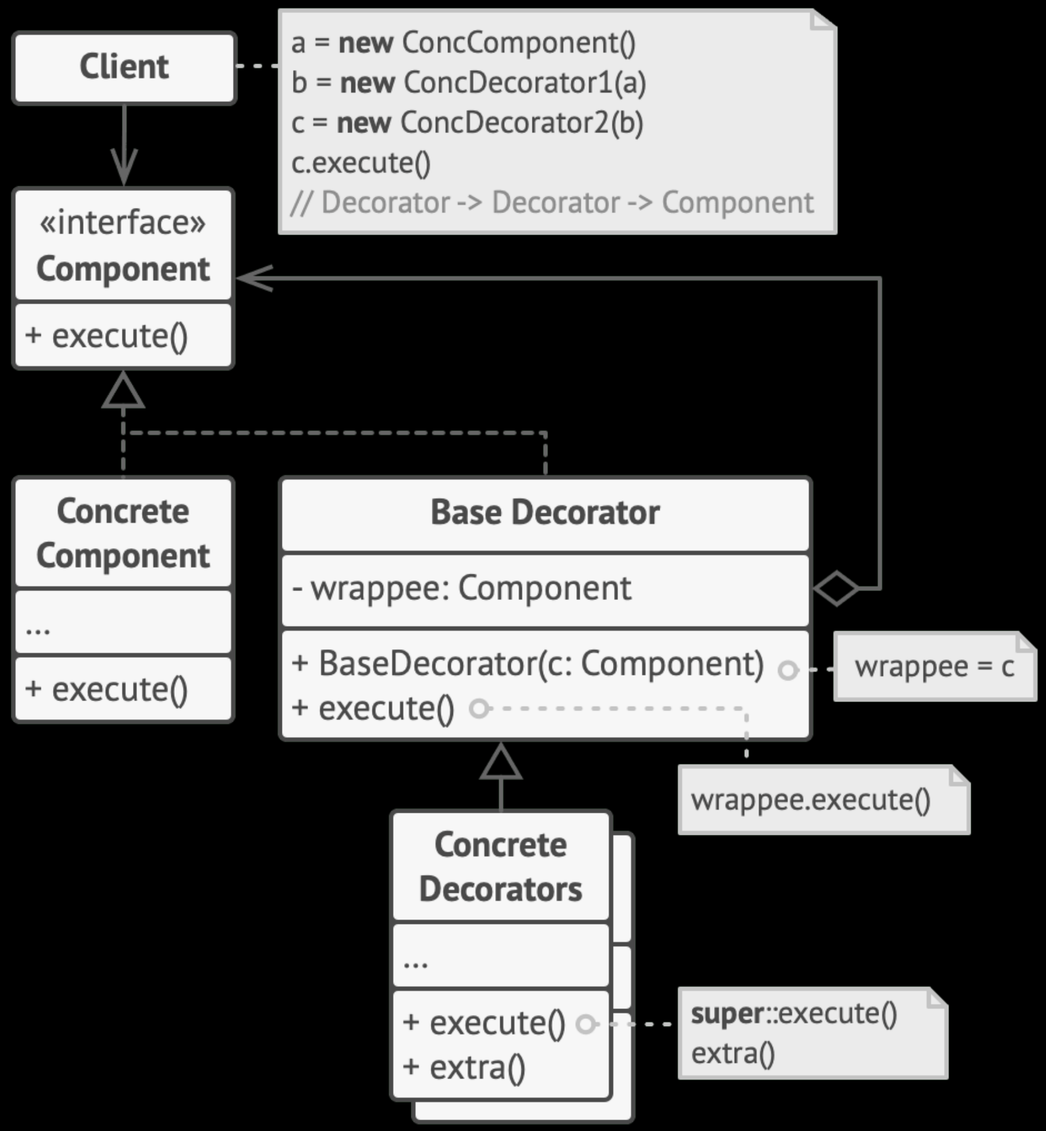
- 기존 서비스를 데코레이터 안에 넣는 방식으로(wrapping) 집합 관계를 만들어서
- 기존 기능에 더불어 추가 기능을 런타임에 제공하는 패턴!
- ex. 로깅 기능, 캐싱 기능 ...!



정의

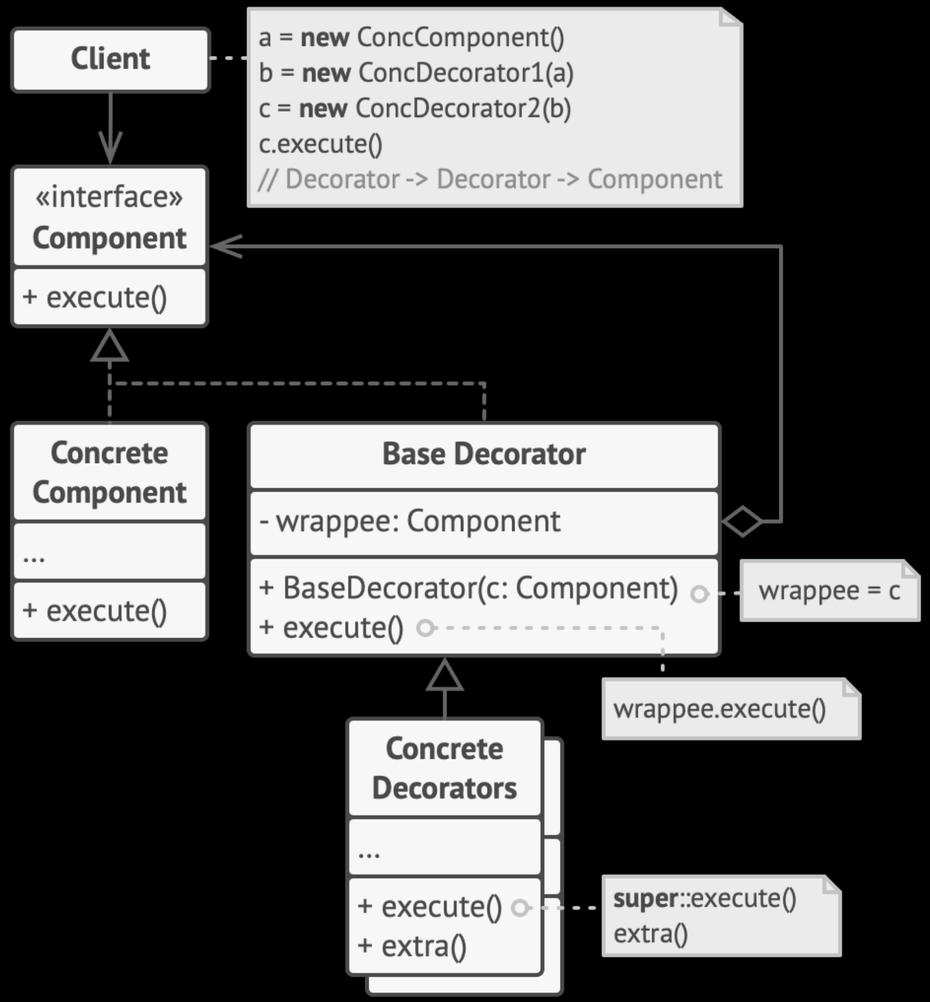


구성 요소


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
«interface»
Component
+ execute()
Concrete
Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()

ko

문제의 해결

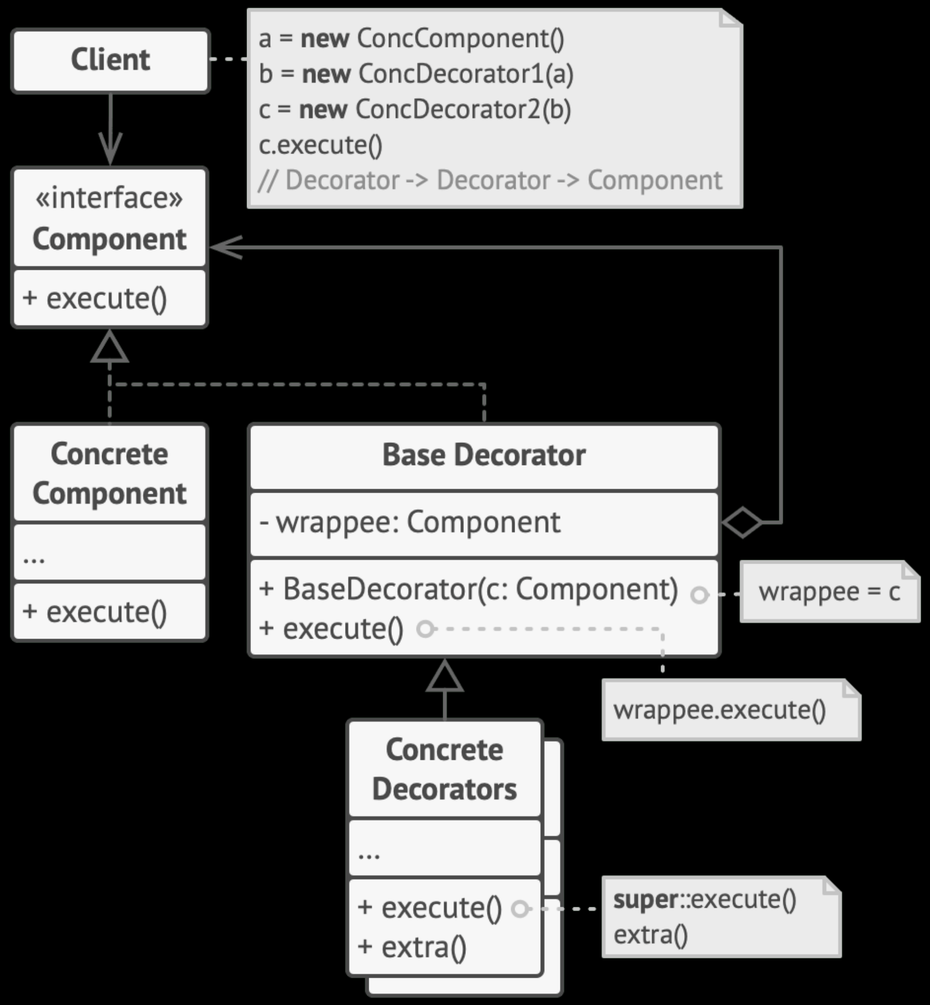



```swift
// MARK: Component(Interface)

protocol 포장할수있는 {
  func 포장풀기()
}
```

또는

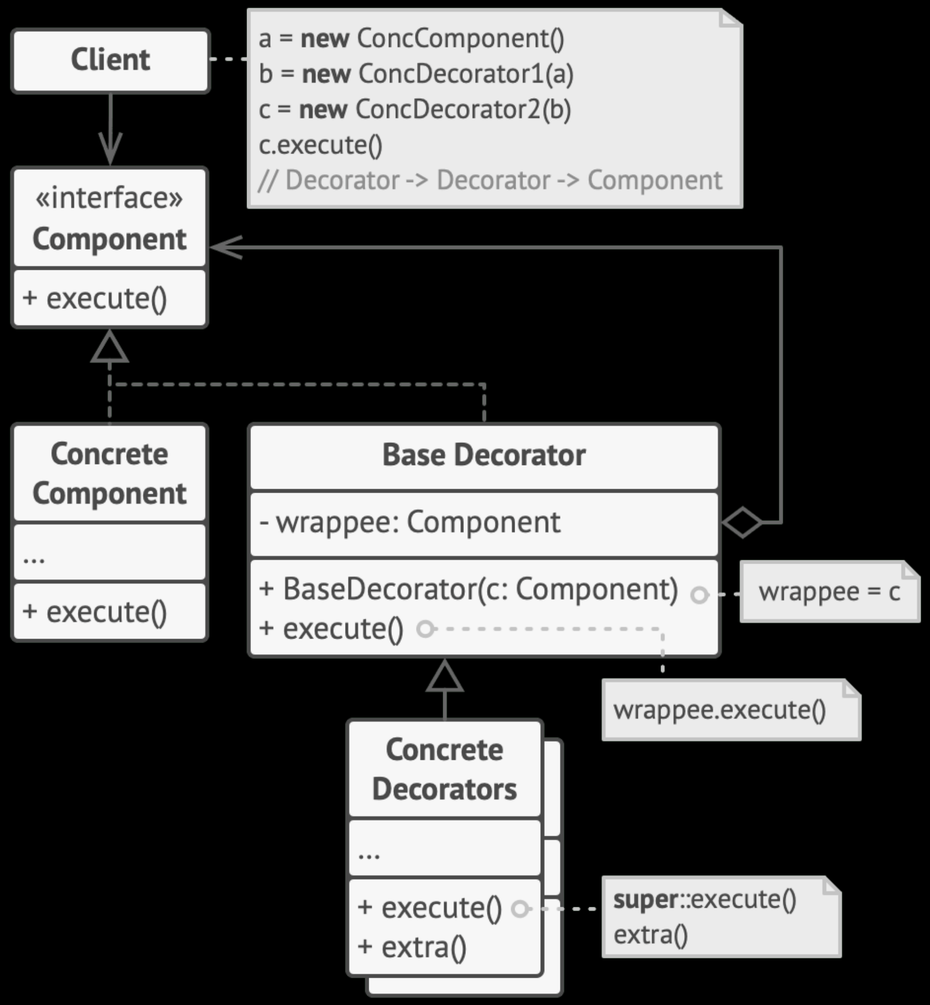
```swift
// MARK: Component(Abstract Class)

class 포장할수있는무언가 {
  func 포장풀기() {

  }
}
```

문제의 해결



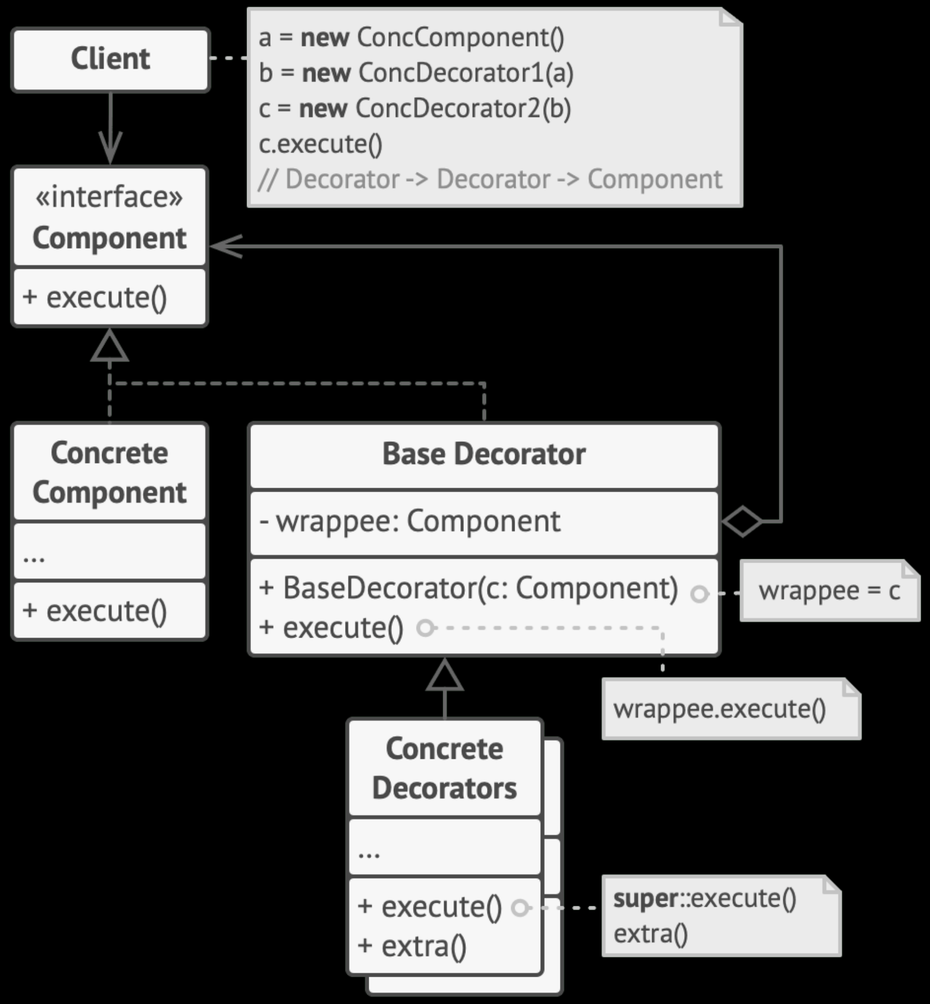

```swift
// MARK: Concrete Component

final class 상품: 포장할수있는 {
  func 포장풀기() {
    print("당근_한_다발", terminator: "\n")
  }
}
```

문제의 해결



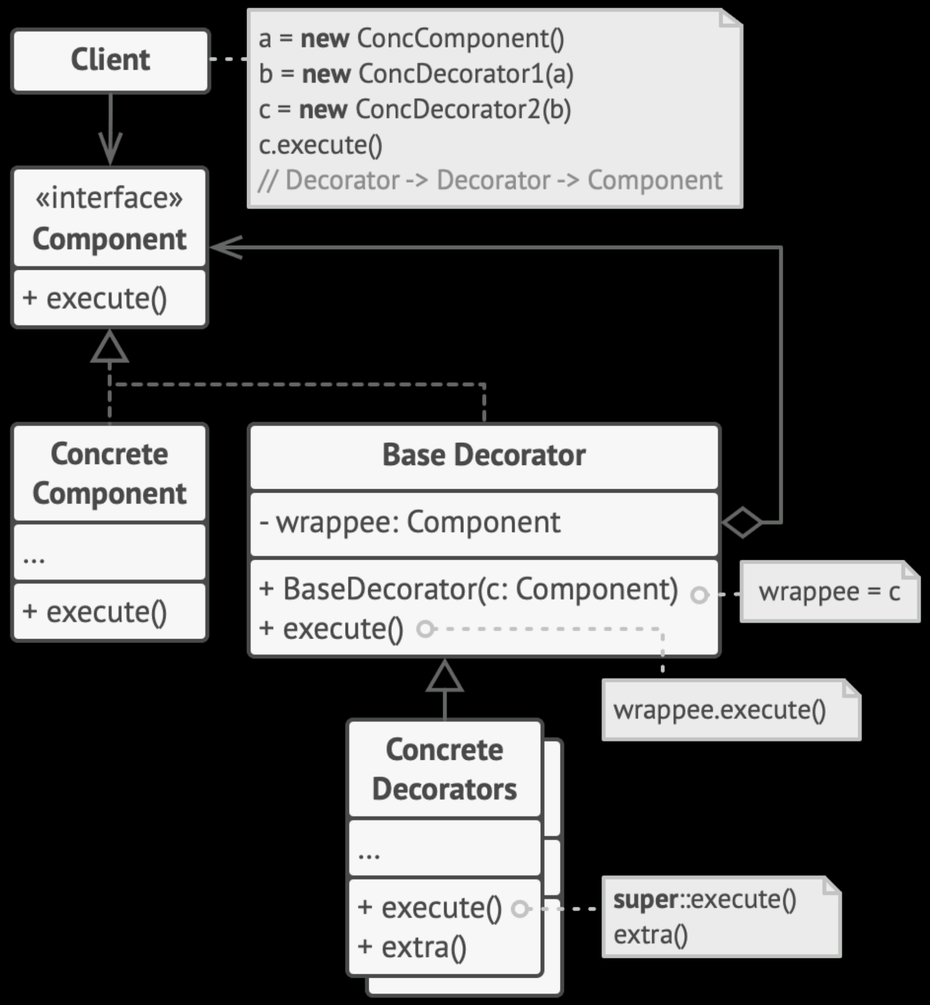

```swift
// MARK: Base Decorator

class 포장: 포장할수있는 {
  private let 포장대상: 포장할수있는

  init(포장대상: 포장할수있는) {
    self.포장대상 = 포장대상
  }

  func 포장풀기() {
    self.포장대상.포장풀기()
  }
}
```

문제의 해결



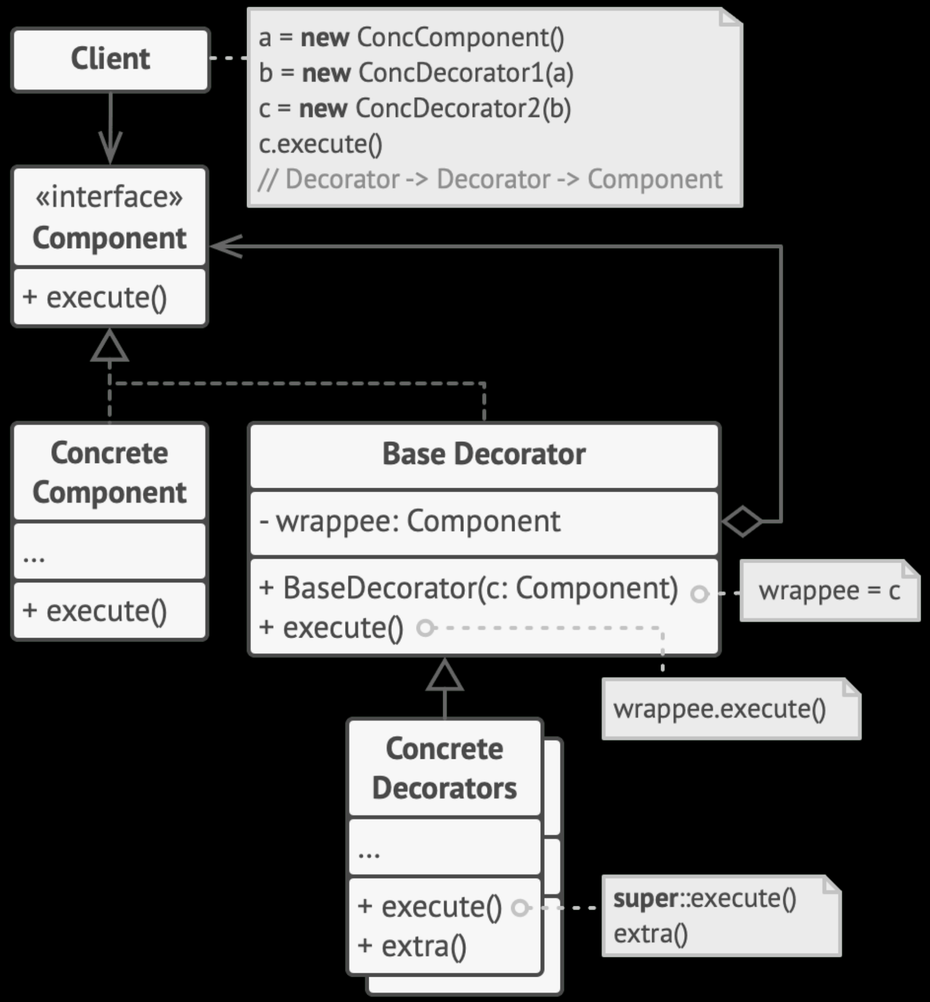

```swift
// MARK: Decorators

final class 비닐포장: 포장 {
  private func 비닐포장소리() {
    print("비닐비닐", terminator: " ")
  }

  override func 포장풀기() {
    비닐포장소리()
    super.포장풀기()
  }
}

final class 종이포장: 포장 {
  private func 종이포장소리() {
    print("종이종이", terminator: " ")
  }

  override func 포장풀기() {
    종이포장소리()
    super.포장풀기()
  }
}

final class 박스포장: 포장 {
  private func 박스포장소리() {
    print("박스박스", terminator: " ")
  }

  override func 포장풀기() {
    박스포장소리()
    super.포장풀기()
  }
}
```

문제의 해결



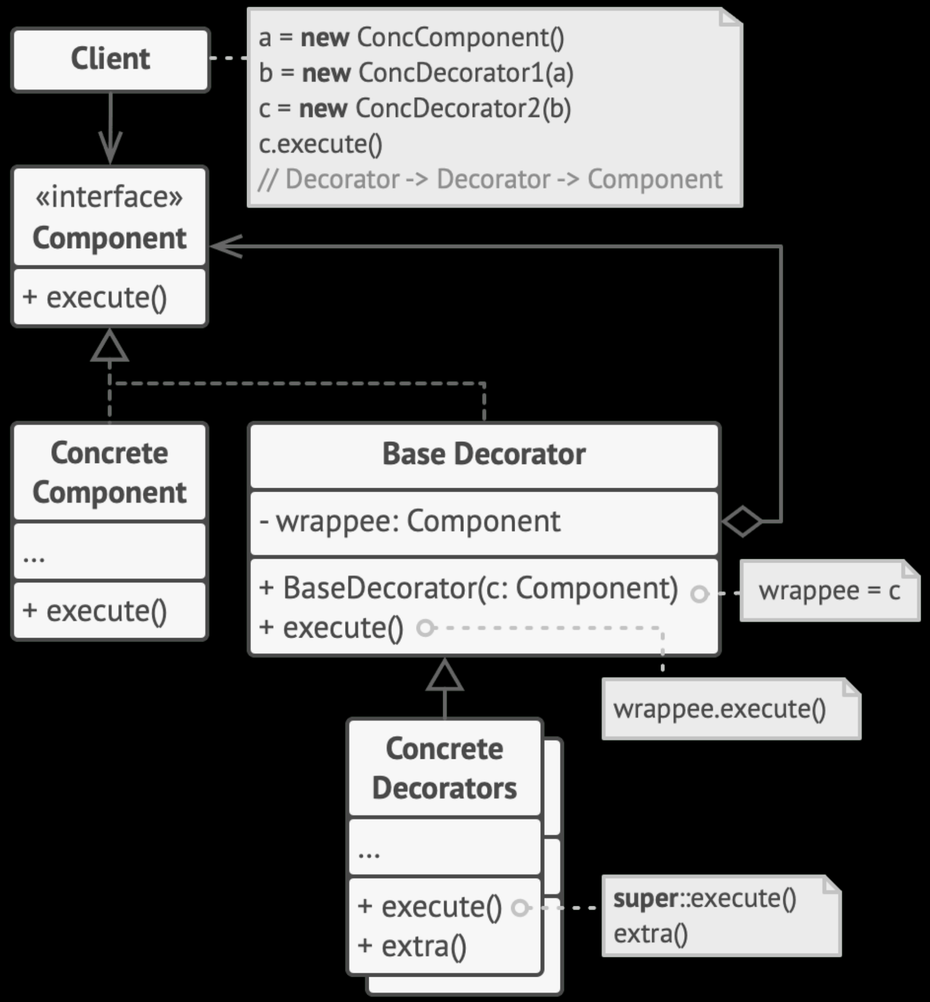

```swift
// MARK: Client

struct 마켓컬리 {
  private var 비닐포장할게요: Bool

  private var 종이포장할게요: Bool

  private var 박스포장할게요: Bool

  init(비닐포장옵션: Bool = true, 종이포장옵션: Bool = true, 박스포장옵션: Bool = true)
{
    self.비닐포장할게요 = 비닐포장옵션
    self.종이포장할게요 = 종이포장옵션
    self.박스포장할게요 = 박스포장옵션
  }

  mutating func 비닐포장옵션끄기() {
    self.비닐포장할게요 = false
  }

  mutating func 종이포장옵션끄기() {
    self.종이포장할게요 = false
  }

  mutating func 박스포장옵션끄기() {
    self.박스포장할게요 = false
  }

  func 제품배송() -> 포장할수있는 {
    var 상품: 포장할수있는 = 상품()

    if self.비닐포장할게요 {
      상품 = 비닐포장(포장대상: 상품)
    }

    if self.종이포장할게요 {
      상품 = 종이포장(포장대상: 상품)
    }

    if self.박스포장할게요 {
      상품 = 박스포장(포장대상: 상품)
    }

    return 상품
  }
}
```

문제의 해결



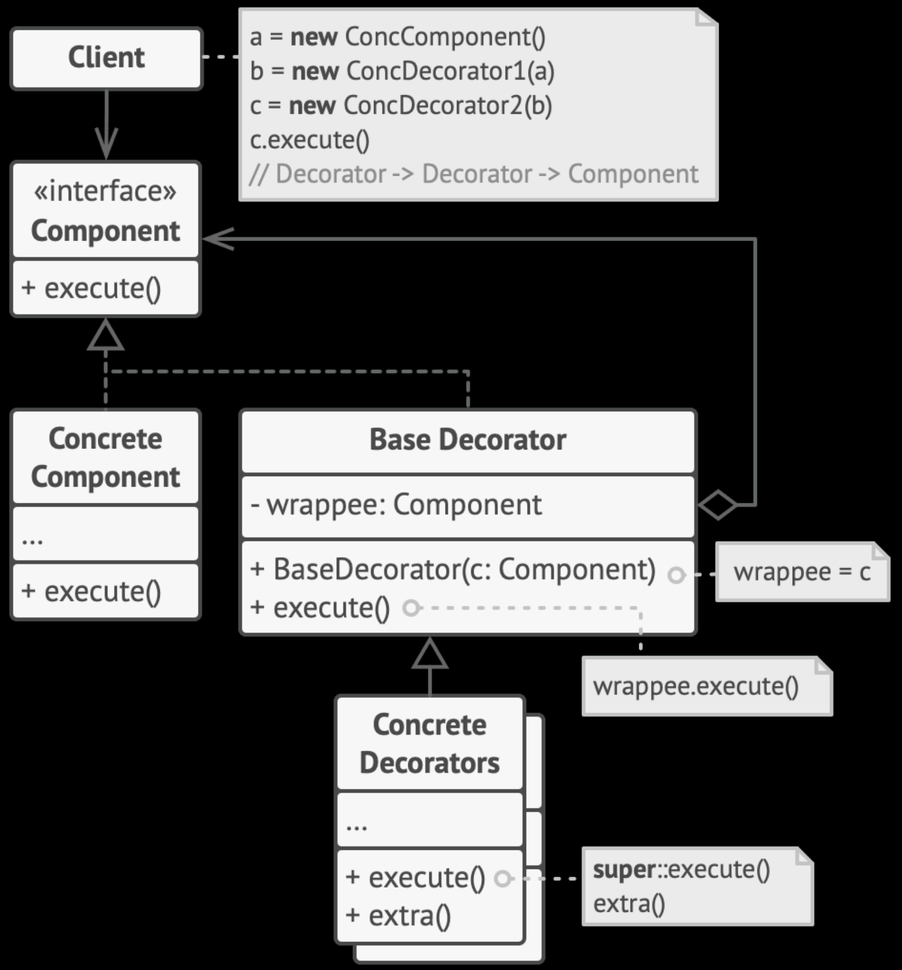

```
func client_code() {
  let 컬리 = 마켓컬리()
  let 오늘의택배 = 컬리.제품배송()
  오늘의택배.포장풀기()
}

// 박스박스 종이종이 비닐비닐 당근_한_다발
```

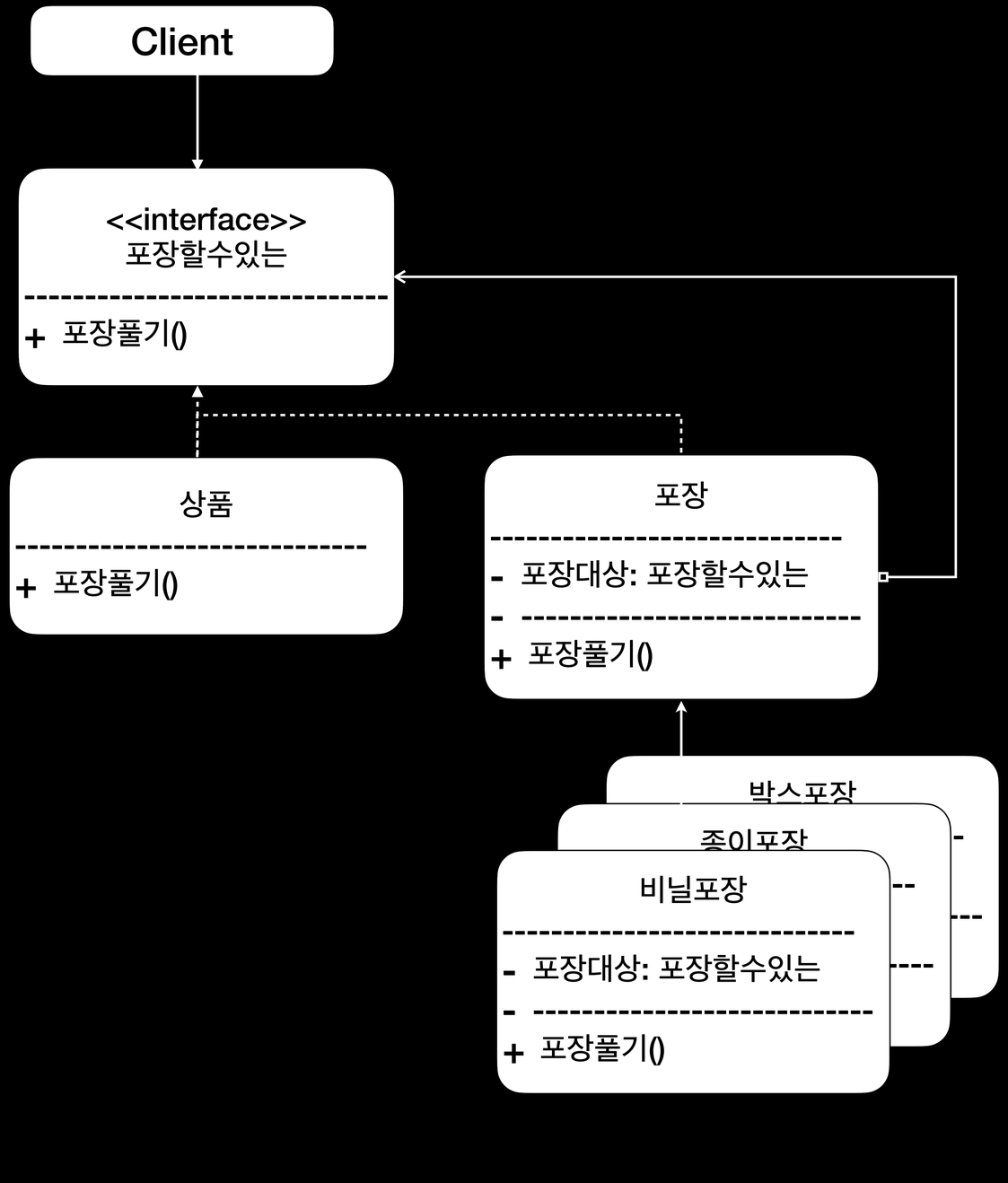
문제의 해결




```
func client_code() {
  var 컬리 = 마켓컬리()
  컬리.비닐포장옵션끄기()
  컬리.박스포장옵션끄기()
  let 오늘의택배 = 컬리.제품배송()
  오늘의택배.포장풀기()
}

// 종이종이 당근_한_다발
```

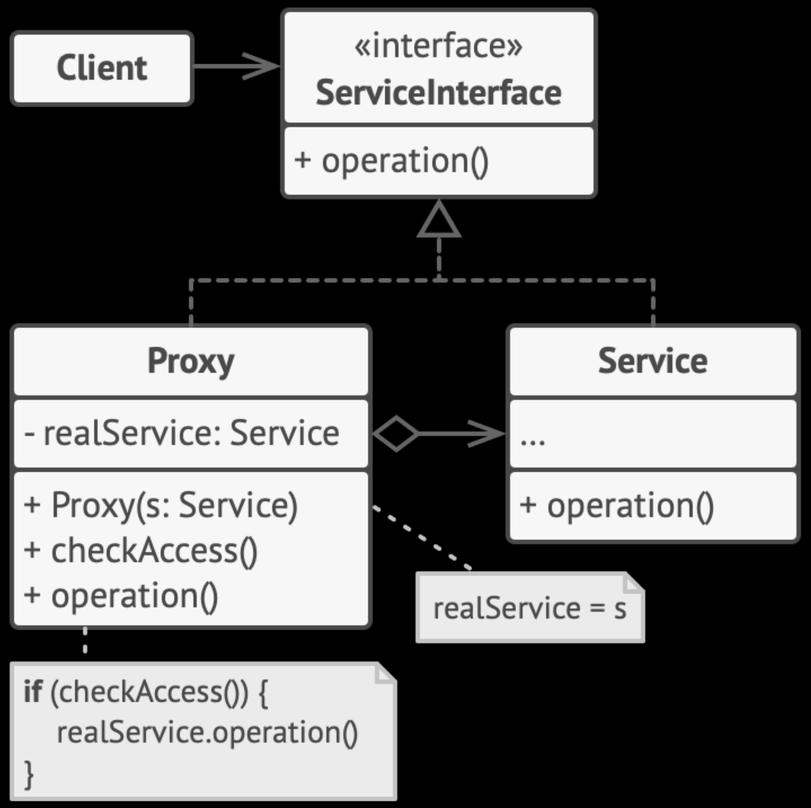
구성 요소


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
«interface»
Component
+ execute()
Concrete
Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()

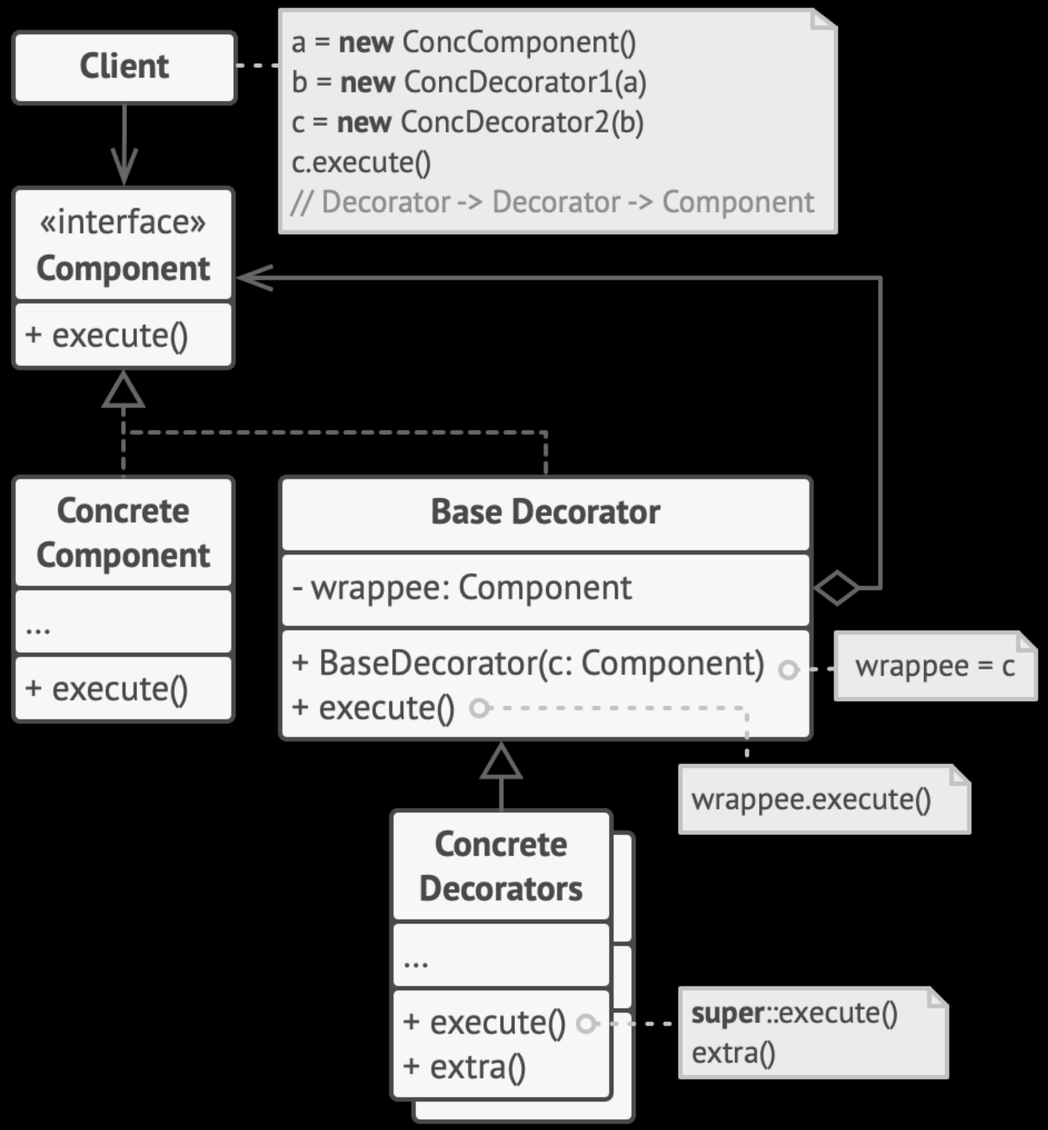

Client
<<interface>>
포장할수있는
+ 포장풀기()
상품
+ 포장풀기()
포장
- 포장대상: 포장할수있는
+ 포장풀기()
비닐포장
- 포장대상: 포장할수있는
+ 포장풀기()

프록시 vs. 데코레이터

- 공통점
  - 원래 객체와 원본 객체의 컨테이너가 같은 인터페이스를 구현한다.
  - 모두 집합 관계에 있는 객체에게 작업을 위임한다.
  - 컨테이너 객체의 사용으로 추가 기능을 제공한다!
- 차이점
  - 패턴의 목적
  - 프록시 - 서비스 대타 / 데코레이터 - 런타임 기능 추가



Proxy



Decorator

장단점과 주의 사항

- 자식 클래스를 만들지 않고 객체의 행동을 확장할 수 있다.
- 런타임에 객체의 임책을 추가할 수 있다.
- 의존성을 주입하기 전에 구현에서 가로채서 역할을 지정할 수 있다.
- SRP(단일 책임 원칙)
  - 각 객체가 다른 책임을 구현하고, 책임을 중첩하는 방식으로 기능을 추가할 수 있다.
  - 다양한 행동들의 여러 변형들을 구현하는 모놀리식 클래스를 여러 개의 작은 클래스들로 나눌 수 있다.
  - 서비스 고유의 로직과 기타 로직을 분리할 수 있다. (횡단 관심사 처리에 유용)

로깅 → 보안

다음 biz..☆ ← 트랜잭션

(핵심) 비즈니스 로직

계좌이체 | 입출금 | 이자계산

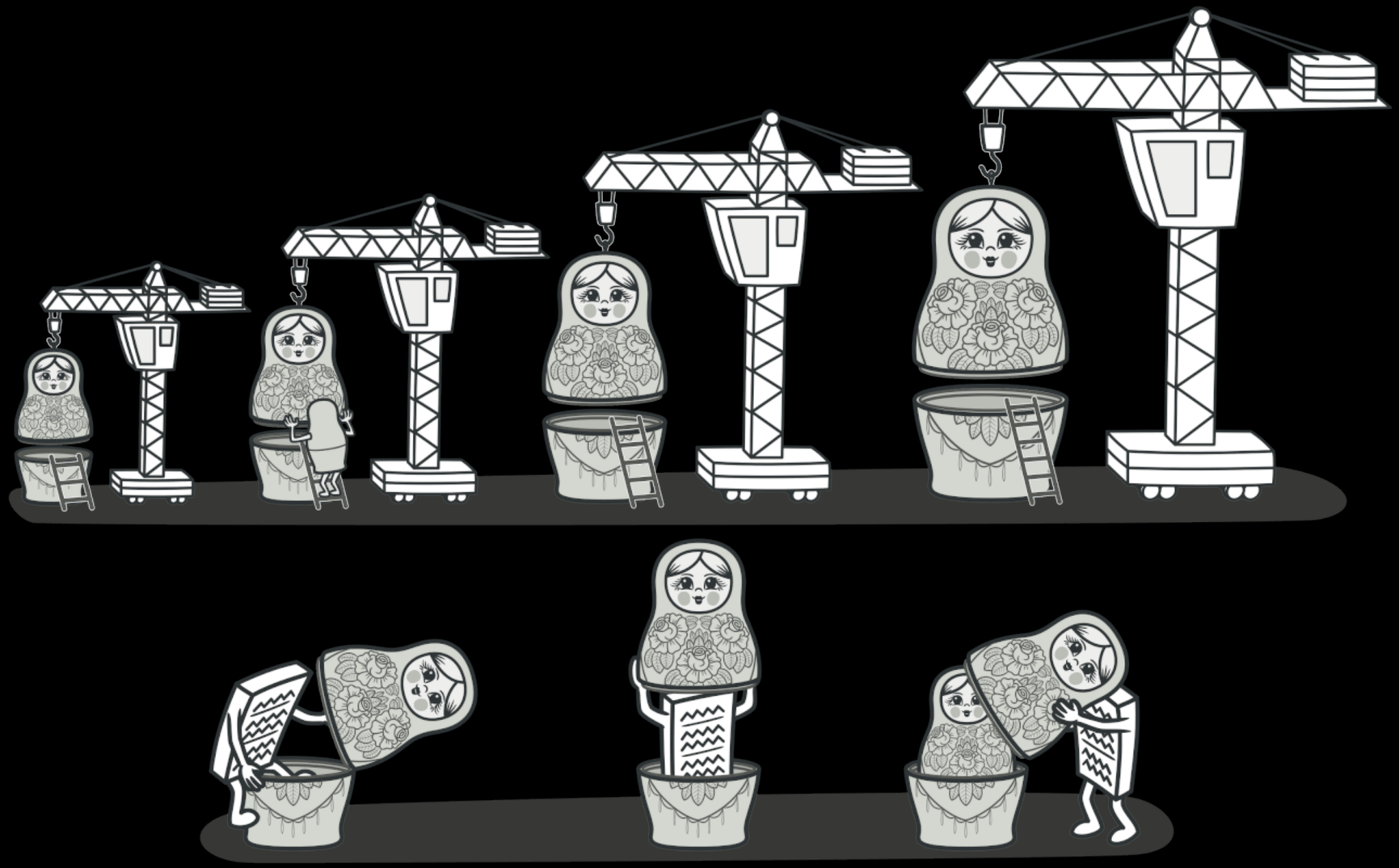
횡단 관심사

- 로깅
- 보안
- 트랜잭션

장단점과 주의 사항

- 래퍼들의 스택에서 특정 래퍼를 제거하기가 (불가능하지는 않지만) 어렵다.
- 데코레이터의 행동이 순서에 매우 의존적이다.
- 계층들의 초기 설정 코드가 보기 흉할 수 있다.

정리



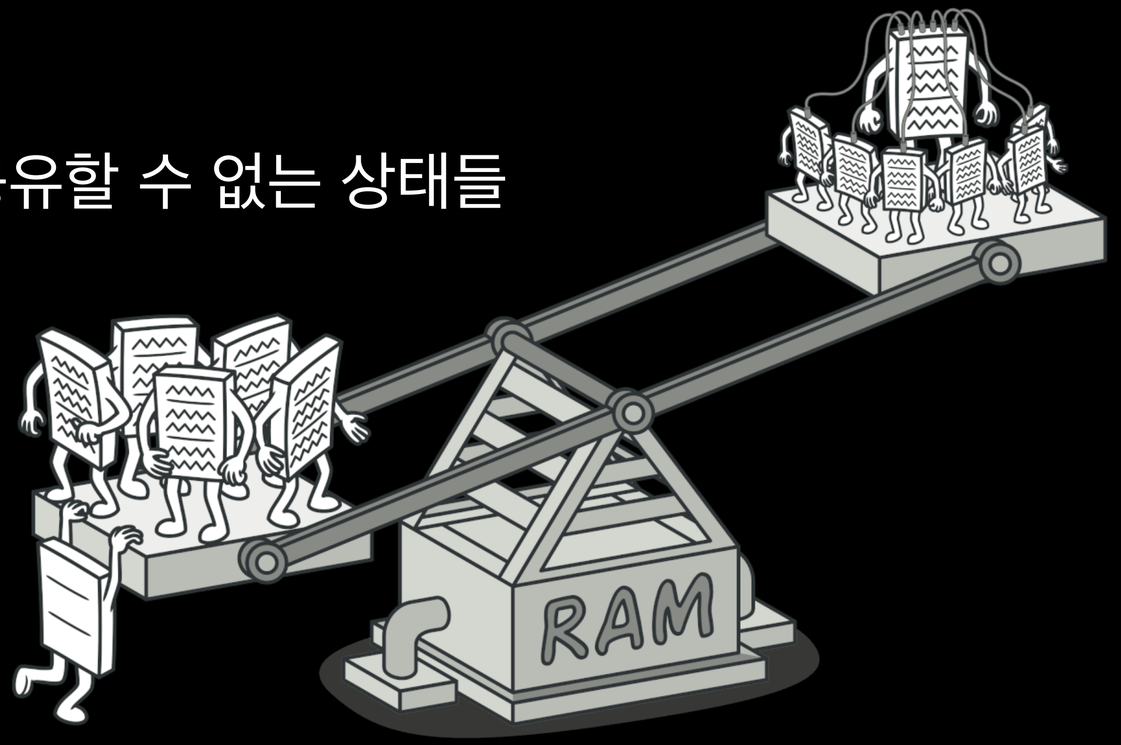
Flyweight

문제

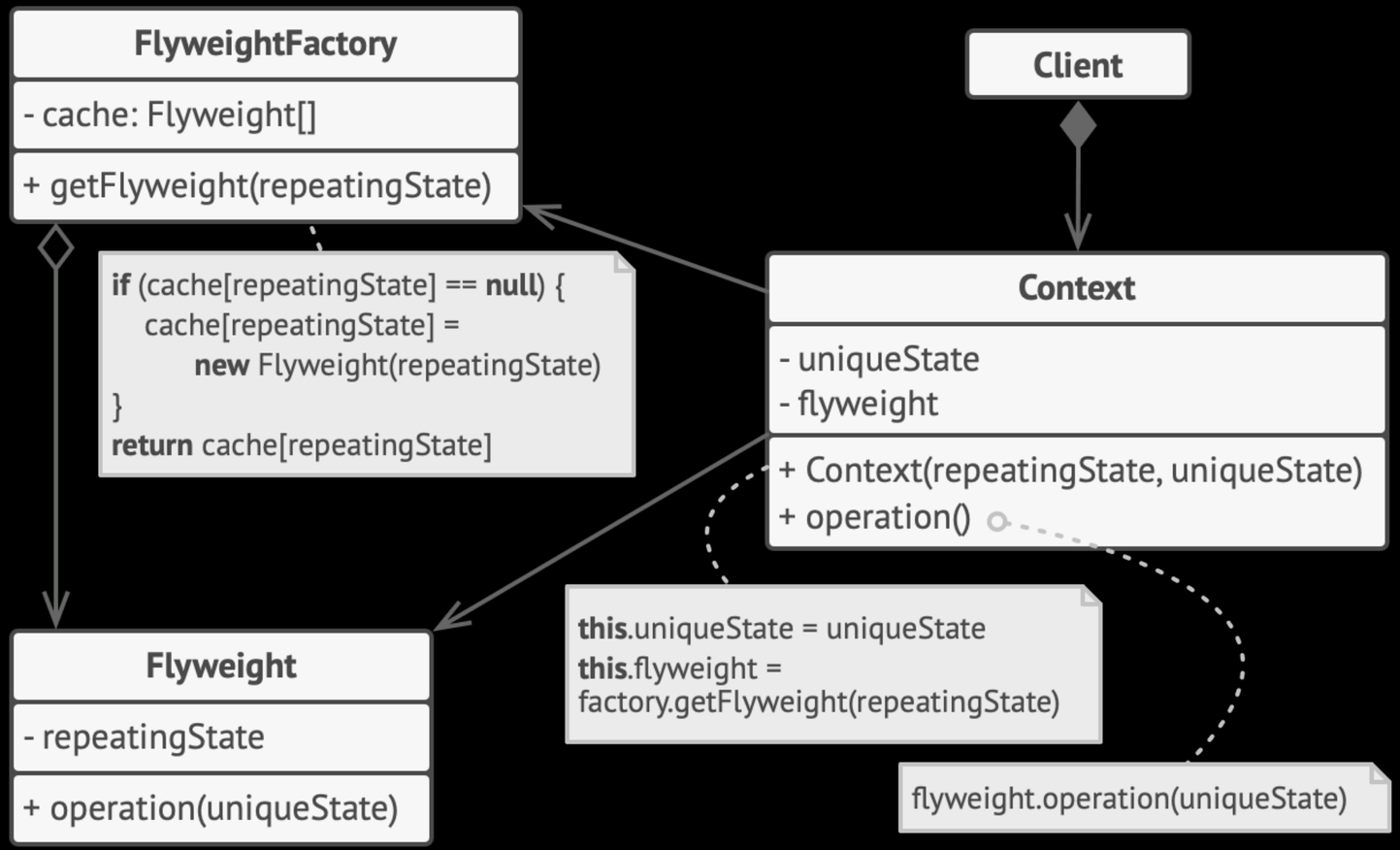
- 객체의 상태 중에서 인스턴스마다 다르지도 않고(vary), 변하지도 않는(immutable) 한 상태들
- 이런 상태들을 가진 인스턴스가 엄청 많이 생성되면 메모리 낭비가 심하다.
- 중복되는 상태들을 인스턴스들이 공유하게 해서 공간을 절약할 수 없을까?

정의

- intrinsic state(immutable, shared)
  - 인스턴스 간에 중복되고, 변하지 않아서 공유되는 것이 나은 상태들
- extrinsic state(mutable, unique)
  - 인스턴스 마다 고유하고, 값이 바뀔 수 있어서 공유할 수 없는 상태들
- intrinsic과 extrinsic을 따로 관리하는 것이



구성 요소

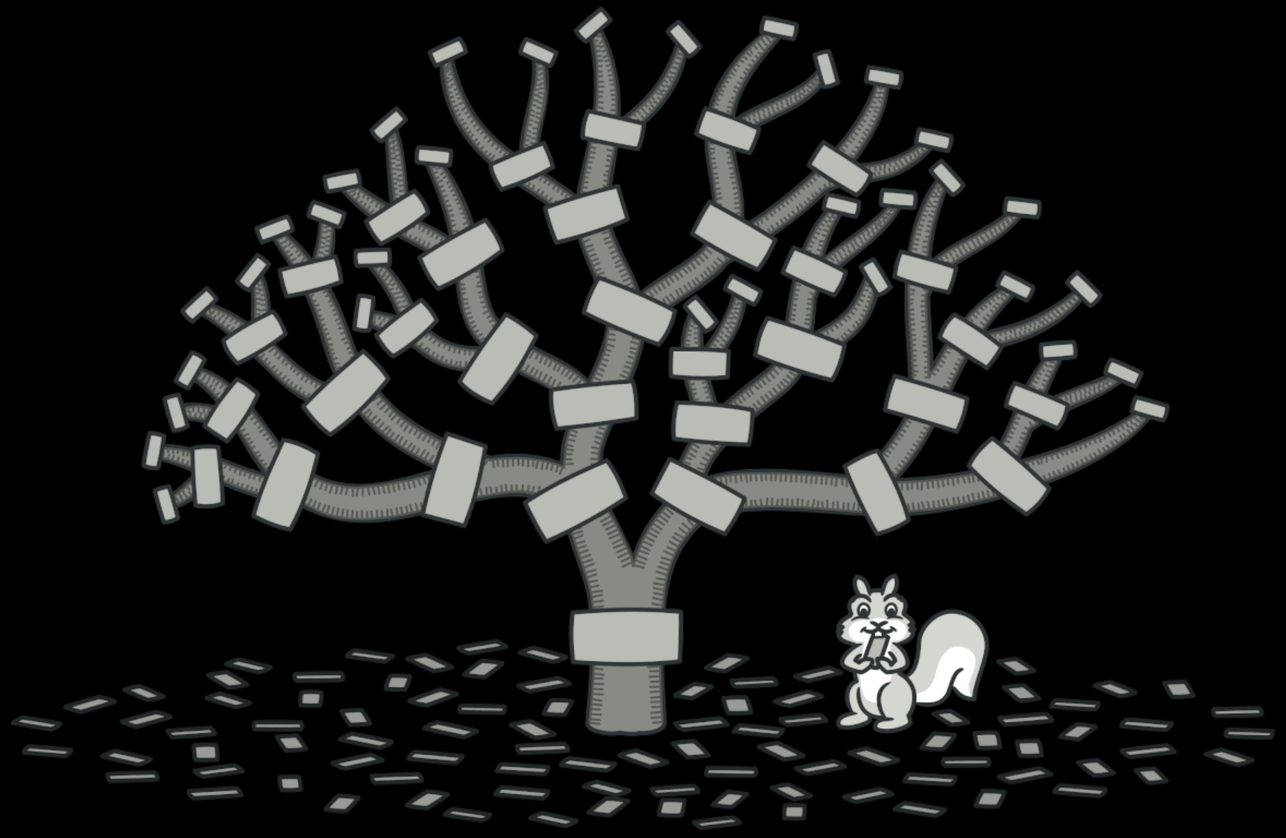

FlyweightFactory
- cache: Flyweight[]
+ getFlyweight(repeatingState)
if (cache[repeatingState] == null) {
cache[repeatingState] =
new Flyweight(repeatingState)
}
return cache[repeatingState]
Client
Context
- uniqueState
- flyweight
+ Context(repeatingState, uniqueState)
+ operation()
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
Flyweight
- repeatingState
+ operation(uniqueState)
flyweight.operation(uniqueState)

장단점과 주의 사항

- 유사한 객체가 많을 때 메모리 공간을 절약할 수 있다.
- RAM 절약 vs. CPU 절약
  - 인스턴스를 만들 때마다 플라이웨이트 메서드를 자주 호출하면....?
- 상태가 분리되어 있으니 객체 구현을 읽기가 어려워진다.

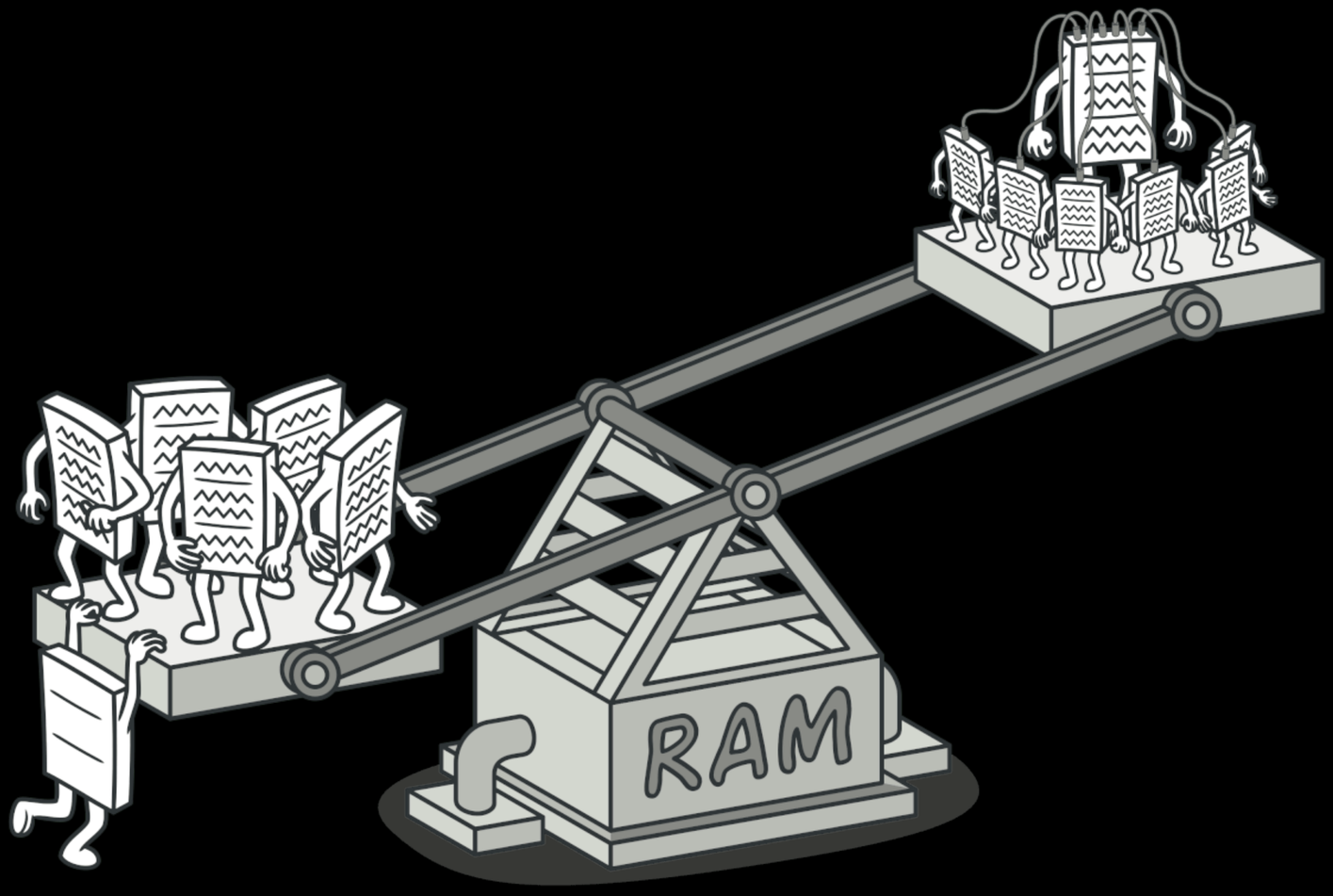
Composit 패턴

- 컴포짓 패턴의 노드들에서 공유되는 부분들을 플라이웨이트 패턴으로 경량화할 수 있다.



정리

- 공간 자원 절약



들어주셔서 감사합니다...