


[디자인패턴] Adapter Pattern

☰ 태그	 Design Pattern
🔗 URL	https://refactoring.guru/ko/design-patterns/adapter
☑ 필기	<input checked="" type="checkbox"/>
🕒 생성 날짜	@2024년 5월 1일 오후 1:38
🕒 최종 편집	@2024년 5월 3일 오후 10:58

어댑터란 ?

실생활 어댑터

개발자 어댑터

어댑터의 필요성

문제

해결책

이론적 사용법

Object Adapter

Class Adapter

예시

구현 방법

사용처

실제 사용

그냥 단순 구현

Adapter Pattern 을 사용하여 개선한 구현

장단점

장점

단점

어댑터란 ?

실생활 어댑터

어학사전

다른 어학정보 7 ▾

[국어사전]

어댑터 (adapter)

1 기계나 기구 따위를 다목적으로 사용하기 위한 보조 기구. 또는 그것을 부착하기 위한 보조 기구.

2 종류 장치의 유출액이 나오는 곳에 대어 다른 장치와 연결하는 화학 실험 기구.

[국어사전 다른 뜻 2](#)

[영어사전]

adaptor

미국·영국 [əˈdæptə(r)]  영국식 

1 (각 별개의 전기 기구를 연결하는 데 쓰는) 어댑터

2 (두 개 이상의 기구를 꽂을 수 있도록 소켓에 연결하는) 어댑터


[영어사전 다른 뜻 2](#)

[어학사전 더보기 →](#)



해외여행용 멀티 어댑터

여기를 눌러 링크를 확인하세요.

 <https://www.ssg.com/item/itemView.ssg?itemId=1000034788483>

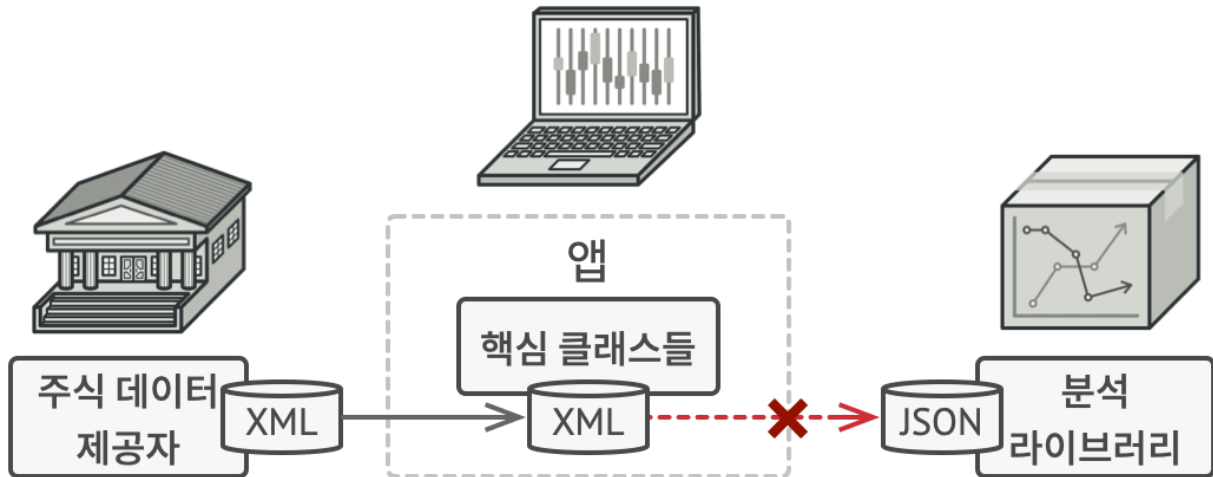
SSG.COM

개발자 어댑터

어댑터는 **호환되지 않는 인터페이스**를 가진 객체들이 **협업할 수 있도록 하는 것**

어댑터의 필요성

문제



(대부분의 예시)

주식 데이터를 **XML** 형태로 받아와서 이를 분석하는 라이브러리를 통해 데이터를 분석하려고 함

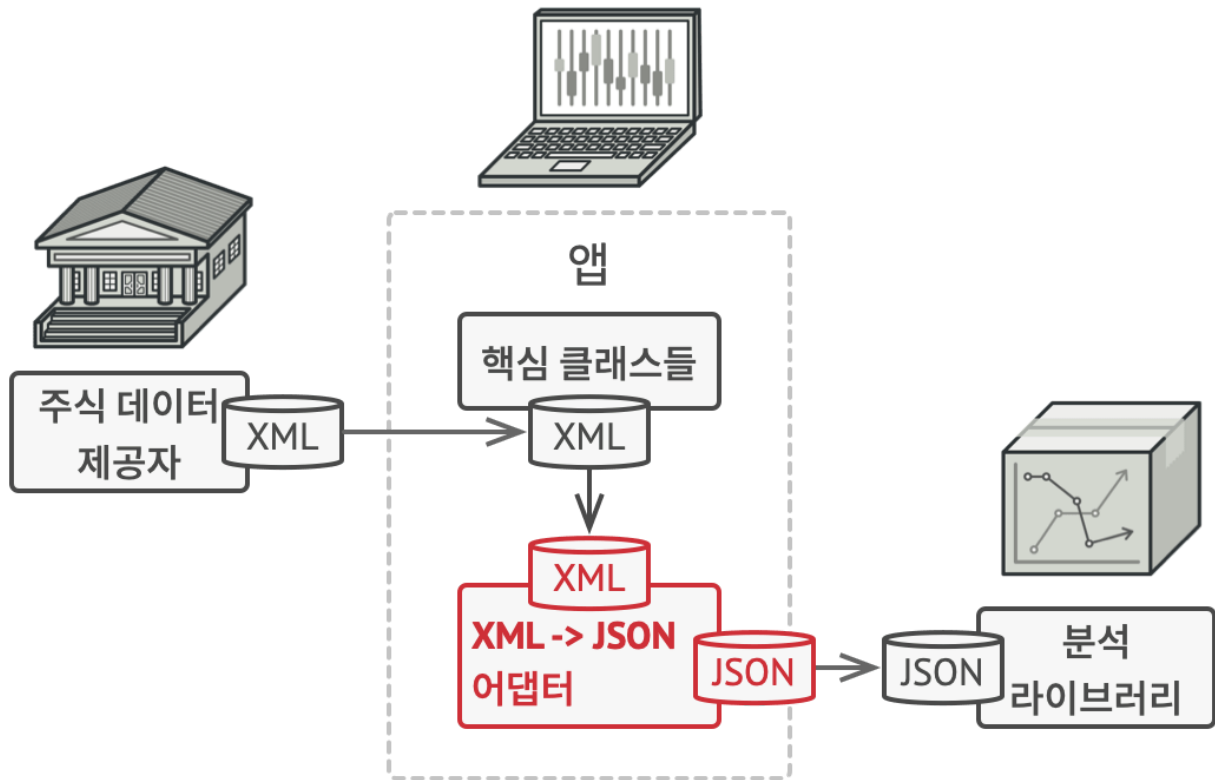
하지만, 이 라이브러리는

JSON 형태만 호환됨 !

우리가 라이브러리를 바꿀 수는 없는데....

어떻게 하면 분석해주는 라이브러리를 **가장 적합하게** 사용할 수 있을까 ?

해결책



JSON 형태를 필요로 하는 라이브러리가 **XML** 형태를 이해할 수 있도록 변환 !

즉, XML을 JSON 형태로 **Wrapping** 해주는 **어댑터** 를 사용

그리고 그렇게 Wrapping 된 데이터를 분석 라이브러리에 전달

이론적 사용법

Object Adapter

▼ Object Composition Principle

Composition is the design technique in object-oriented programming to implement **has-a** relationship between

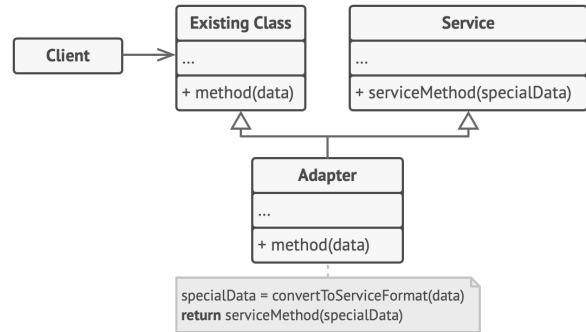
Class Adapter

▼ Inheritance

상속을 이용한 방법
단, 두 가지 클래스를 한 번에 상속할 수 있는 언어만 사용 가능
(Swift에서는 프로토콜로 가능할 듯?)

objects. Composition in java is achieved by using instance variables of other objects.

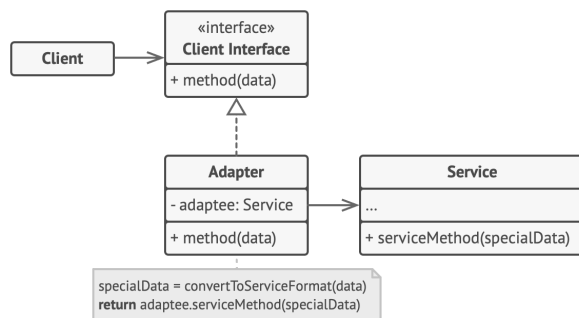
Composition 이란 OOP에서 **has-a** 관계를 구축하기 위한 방법
다른 객체의 인스턴스를 갖고 있는 것
결국 그냥 객체 내부에 다른 객체의 인스턴스를 갖도록 하는 것



Composition vs Inheritance | DigitalOcean

Technical tutorials, Q&A, events — This is an inclusive place where developers can find or lend support and discover new ways to

<https://www.digitalocean.com/community/tutorials/composition-vs-inheritance>

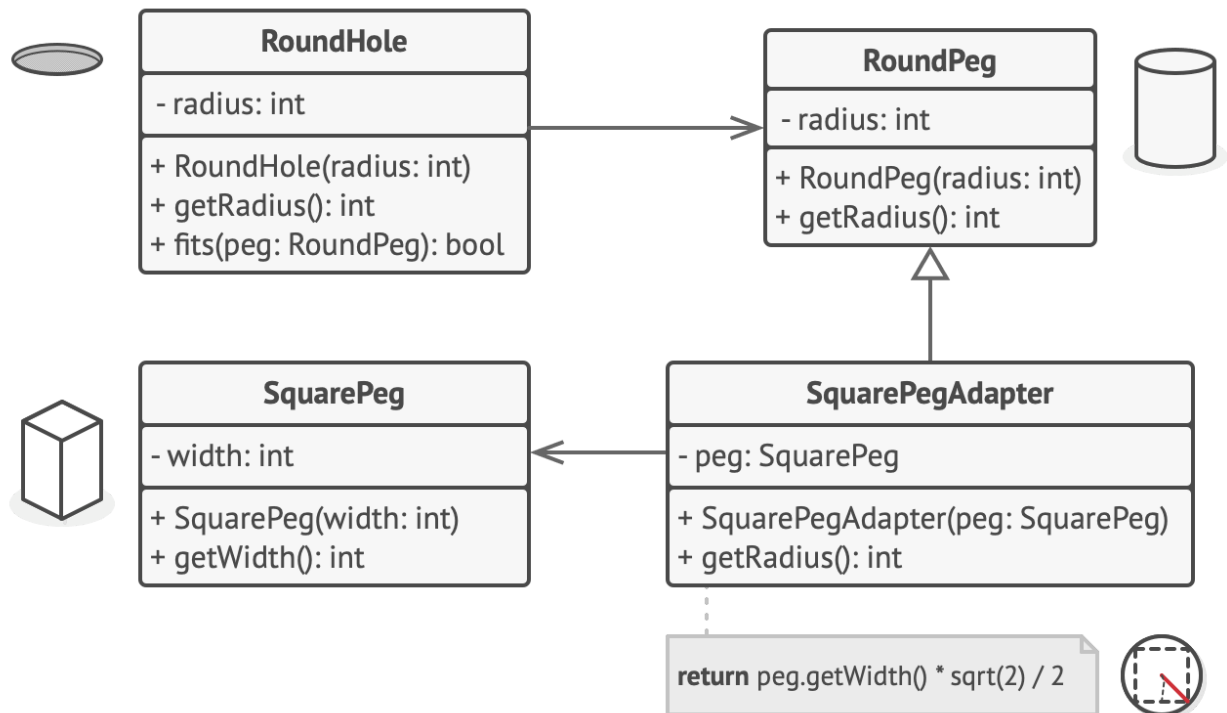


- **Client** - 클라이언트 ... **Client Interface** 의 구현체를 갖고 있음
- **Client Interface** - 클라이언트에 대한 인터페이스 ...
- **Adapter** - **Client Interface** 의 data를 **Service** 에서 사용할 수 있도록 변환
- **Service** - **Client** 에서 호환되지 않기 때문에 직접적으로 사용할 수 없는 서비스

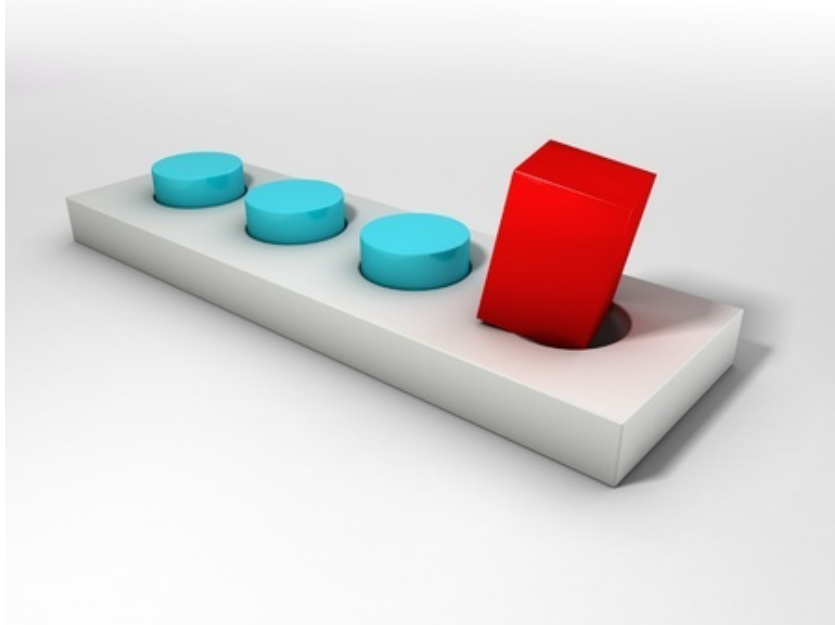
1. **Client** 는 **Service** 를 사용하고 싶어 !
2. 어 ... 근데 **Service** 를 쓰기에는 데이터 형태가 조금 다르네 ... 어떻게 쓸까 ...?

3. 아 ! **Adapter** 한테 **Service** 쓸 수 있도록 데이터 형태 바꿔달라고 해야겠다 !
4. 야야 **Adapter** ! 너 내가 **Service** 써야하는데 ... 내가 코드 바꾸기는 귀찮으니까 !
 내 **Interface** 그대로 받아와서 구현하는데, 내가 주는 데이터 그대로 변환해서 **Service** 써줘 !

예시



▼ 틈새 영어 설명) Square Peg in a Round Hole



<https://careerpivot.com/2018/are-you-a-square-peg-trying-to-fill-a-round-hole/>

마치 ... 34기 스터디에 참여하는 ... 32기 나처럼 ...

정사각기둥이 원형 구멍을 지나갈 수 있을까 ??

1. 이를 판단하기 위해서는, 정사각기둥에 딱 맞는 원형 기둥을 만들어야 함 !
2. `SquarePegAdapter` 에서 기존 `SquarePeg` 를 인스턴스로 갖기 위해 입력값으로 주입
3. `SquarePegAdapter` 는 `RoundPeg` 를 따라야지 `RoundHole` 에 들어갈 수 있는지 판단 가능
이때, `RoundPeg`의 `getRadius()` 를 `SquarePeg` 의 `width` 로부터 구현
4. 그러면, `RoundHole` 에 `fit(peg:)` 한지 확인 가능 !

생성자는 과감히 생략 ...

```
class RoundHole {  
    private var radius: Int  
    func fits(_ peg: RoundPeg) -> Bool {  
        return self.radius >= peg.getRadius()  
    }  
}
```

```
protocol RoundPeg {
    func getRadius() -> Int
}
```

```
protocol SquarePeg {
    func getWidth() -> Int
}
```

```
class SquarePegAdapter: RoundPeg {
    private let squarePeg: SquarePeg

    func getRadius() -> Int { // 여기도 그냥 Int 형으로 할게요..
        return squarePeg.getWidth() / 2 * sqrt(2)
    }
}
```

사용은 ?

```
let hole: RoundHole = .init(radius: 5)
let roundPeg: RoundPeg = RoundPegImpl(radius: 5)
hole.fits(roundPeg) // true

let squarePeg: SquarePeg = SquarePegImpl(width: 10)
hole.fits(squarePeg) // 컴파일 에러 !

let squarePegAdapter: SquarePegAdapter = .init(peg: squarePeg)
hole.fits(squarePegAdapter) // false
```

구현 방법

0. 서로 호환이 안되는 **두 개 이상의 인터페이스가 존재** 해야 함

- **변경을 할 수 없는** 유용한 기능을 가진 어떤 Service
- 이런 서비스를 사용해야 하는 클라이언트(사용처) 한 개 이상

1. 서비스를 사용할 **Client Interface** 를 만들고, Service를 어떻게 사용할지 구성
(인터페이스이기 때문에 구현체는 아직은 X)

2. `Client Interface` 를 따르는 `Adapter` 구현 (아직 함수 내부는 비워두기)
3. `Adapter` 에 사용할 `Service` 를 프로퍼티로 갖고 있도록 만들기
4. `Client Interface` 에서 구현해야 할 코드들을 `Adapter` 에 `Service`를 사용 하면서 적절히 구현 !
5. `Client` 에서는 `Client Interface` 를 따르는 `Adapter` 를 통해 `Service` 의 기능들을 사용 !

사용처

- 기존 클래스를 사용하고 싶지만, 다른 코드와 호환이 되지 않을 때 사용
- 자식 클래스들에 공통적으로 필요한 기능이 있는데, 부모 함수에 넣을 수 없을 때 사용
 - 어댑터에 필요한 함수를 넣어서 사용하면 됨 !

실제 사용

구글 로그인을 추가하는데, 실제로 사용할 데이터 형식이 다름 !
구글 로그인 서비스를
`wrapping` 하는 방식을 사용해보자 !

그냥 단순 구현

```
struct GoogleUser {
    var email: String
    var password: String
    var token: String
}

class GoogleAuthenticator {
    func login(
        email: String,
        password: String,
        completion: @escaping (GoogleUser?, Error?) -> Void
    ) {
        // 네트워크 통신을 통해 토큰이랑 결과값 등이 돌아옴
        let token = "special-auth-token"
```

```

        let user = GoogleUser(email: email, password: password)
        completion(user, nil)
    }
}

```

- **대충** 이렇게 구현되어 있는 서드 파티 클래스

```

struct User {
    let email: String
    let password: String
}

struct Token {
    let value: String
}

```

```

final class LoginViewController: UIViewController {
    var authService: GoogleAuthenticator = .init()

    private var emailTextField = UITextField()
    private var passwordTextField = UITextField()

    private func login() {
        guard let email = emailTextField.text,
              let password = passwordTextField.text
        else { return }

        authService.login(
            email: email,
            password: password
        ) { (googleUser, error) in
            print("googleUser email: \(googleUser.email)")
            print("googleUser password: \(googleUser.password)")
            print("googleUser token: \(googleUser.token)")

            // 우리는 User 타입으로 사용하고 싶음

```

```

        let user = User(email: googleUser.email, password: googleUser.password)
        let token = Token(value: googleUser.token)
        // 추가 코드
    }
}
}

```

- 이렇게 구현 가능 !
 - 하지만 .. 여기에 만약 다른 Authentication Service가 필요하다면 ?
 - 그냥 인스턴스로 또 넣으면 되지 !
 - 그러다 변동사항 생기면 ..?
 - 고쳐.. 야지..?
 - 그렇게 하는것보다 인터페이스를 갖고 있는게 변화도 적고 더 좋지 않을까..?
 - 쓰고자 하는 데이터 타입이 다르다면 ?
 - 그냥 함수 내에서 데이터 매핑 해주면 안돼 ?
 - 데이터 매핑이라는 다른 책임을 갖게 되는데 ..?
 - 중간에 어떤 다른 동작을 해야한다면 ?
 - 이것도 요 ViewController 의 함수 내에서 하게 되면 다른 책임을 또 갖게 되는데 ?

Adapter Pattern 을 사용하여 개선한 구현

```

protocol AuthService {
    func login(
        email: String,
        password: String,
        success: @escaping (User, Token) -> Void,
        failure: @escaping (Error?) -> Void
    )
}

```

```

final class GoogleAuthServiceAdapter: AuthService {
    private let authenticator = GoogleAuthenticator()

    func login(
        email: String,

```

```

        password: String,
        success: @escaping (User, Token) -> Void,
        failure: @escaping (Error?) -> Void
    ) {
        authenticator.login(email: email, password: password) {
            guard let googleUser else {
                failure(error)
                return
            }

            let user = User(email: googleUser.email, password: googleUser.password)
            let token = Token(value: googleUser.token)
            success(user, token)
        }
    }
}

```

```

final class LoginViewController: UIViewController {
    var authService: AuthService!

    private var emailTextField = UITextField()
    private var passwordTextField = UITextField()

    public class func instance(with service: AuthService) -> LoginViewController {
        let viewController = LoginViewController()
        viewController.authService = service
        return viewController
    }

    private func login() {
        guard let email = emailTextField.text,
              let password = passwordTextField.text
        else { return }

        authService.login(
            email: email,
            password: password,
            success: { user, token in

```

```

        print("Auth Success: \(user), \(token)")
    },
    failure: { error in
        print("Auth failed: \(String(describing: error))")
    }
}
)
}
}

```

- 이렇게 만들면, 중간에 다른 데이터 형태로 매핑해야하는 문제 사라짐 !
- 또, 다른 Auth Service가 생겨도 `AuthService protocol` 로 래핑하여 손쉽게 사용 가능 !

장단점

장점

- `Single Responsibility Principle`
변환하는 과정을 Adapter에게 맡겨 책임 분리
- `Open Closed Principle`
기존 Client 코드를 변형 없이 사용할 수 있음

단점

- `전반적인 코드 복잡도 증가`
인터페이스랑 구현체 등 몇 가지 새로운 요소들을 구현해야 하기 때문

어댑터 패턴

어댑터는 호환되지 않는 인터페이스를 가진 객체들이 협업할 수 있도록 하는 구조적 디자인 패턴입니다.

 <https://refactoring.guru/ko/design-patterns/adapter>

[iOS - Swift] 5. 디자인 패턴 (구조 패턴) - 어댑터 패턴 (Adapter, Wrapper)

어댑터 패턴 (Adapter, Wrapper) 현재 A 기능을 사용중일때 이와 유사한 B, C 기능이 계속 생겨나면서, B, C 인터페이스도 A와 동일하게 만드는 것을 목적으로 A의 프로토콜을 준수하는 BAdapter, CAdapter로 만들어서 사용하는쪽에서 코드의 변경을 최소화 하

📄 <https://ios-development.tistory.com/1235>



[Swift] Adapter Pattern

Adapter Pattern ✅ Adapter Pattern 아래의 문서를 구입하여 영어 문서를 번역하고 이해한 것을 바탕으로 글을 작성하고 있습니다.

<https://www.raywenderlich.com/books/design-patterns-by->

📄 <https://rldd.tistory.com/404>

