

# 옵저버 패턴 (Observer Pattern)

*Wikipedia의 Observer pattern 문서를 한글로 번역한 뒤 스터디 발표를 위해 정리한 문서입니다.*

👉 [https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern)

## 개요

옵저버 패턴(Observer Pattern)은 객체 사이의 일대다(one-to-many) 의존성을 정의하는 Behavioral 소프트웨어 디자인 패턴의 종류 중 하나입니다.

이 패턴에서는 "subject"라고 불리는 객체가 "observer"들의 목록을 유지하며, 상태가 변경될 때마다 모든 observer들에게 알림을 보냅니다.

이 패턴은 Gang of Four의 "Design Patterns" 책에서 소개되었으며, 이벤트 기반 프로그래밍과 GUI 프레임워크에서 널리 사용되는 디자인 패턴 중 하나입니다.

## 주요 특징

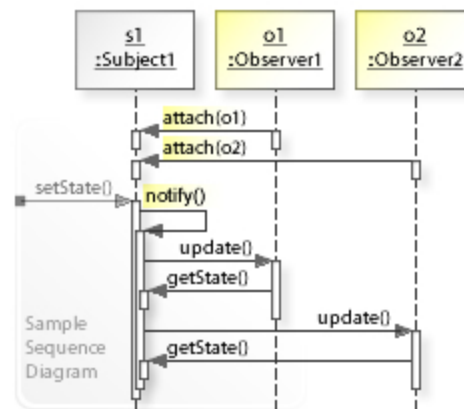
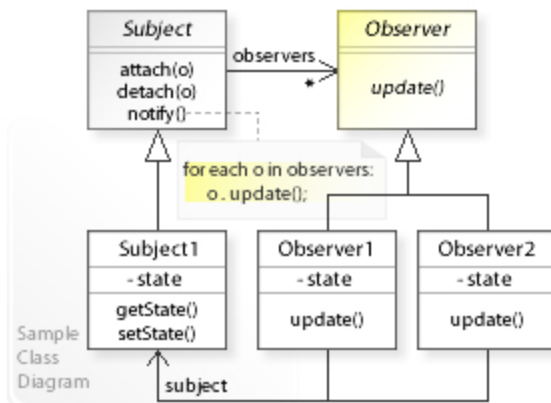
- 일대다 의존성 🖐️ 하나의 subject가 여러 observer들을 관리
- 자동 알림 🖐️ subject의 상태 변경 시 자동으로 observer들에게 통지
- 느슨한 결합 🖐️ subject와 observer 간 인터페이스를 통한 약한 결합
- 동적 관계 🖐️ 런타임에 observer 등록/해제 가능
- 동기적 통신 🖐️ 일반적으로 직접적이고 동기적인 통신 방식

## 핵심 구조

옵저버 패턴은 다음과 같은 주요 구성 요소로 이루어져 있습니다:

1. **Subject (주체)**: observer들의 목록을 유지하고 관리
2. **Observer (관찰자)**: subject의 변경을 감지하기 위한 인터페이스
3. **ConcreteSubject**: 구체적인 상태를 유지하며 변경 시 observer들에게 알림
4. **ConcreteObserver**: ConcreteSubject의 상태를 추적하고 업데이트 받음

## 옵저버 패턴의 UML 구조



## 핵심 요구사항

옵저버 패턴은 다음과 같은 요구사항을 충족합니다:

1. 객체 간 긴밀한 결합 없이 의존성 정의
2. 한 객체의 상태가 변경될 때 의존 객체들을 자동으로 업데이트
3. 하나의 객체가 여러 다른 객체들에게 알림을 보낼 수 있음
4. 런타임에 관찰 관계를 동적으로 추가/제거 가능

## Java 예제: Observer 인터페이스

```
// Observer 인터페이스  
public interface Observer {  
    void update(String message);  
}
```

## Java 예제: Subject 인터페이스

```
// Subject 인터페이스
public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}
```



# Java 예제: ConcreteSubject 구현

```
import java.util.ArrayList;
import java.util.List;

/**
 * ConcreteSubject - 실제 상태를 유지하고 observer들을 관리
 */
public class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }

    public void setNews(String news) {
        this.news = news;
        notifyObservers(); // 상태 변경 시 자동으로 알림
    }
}
```

## Java 예제: ConcreteObserver 구현

```
/**
 * ConcreteObserver - 실제 업데이트를 받아 처리
 */
public class NewsChannel implements Observer {
    private String name;
    private String news;

    public NewsChannel(String name) {
        this.name = name;
    }

    @Override
    public void update(String news) {
        this.news = news;
        System.out.println(name + " received news: " + news);
    }

    public String getNews() {
        return news;
    }
}
```

# Java 예제: 클라이언트 코드

```
public class Main {  
    public static void main(String[] args) {  
        // Subject 생성  
        NewsAgency agency = new NewsAgency();  
  
        // Observer들 생성 및 등록  
        NewsChannel channel1 = new NewsChannel("Channel 1");  
        NewsChannel channel2 = new NewsChannel("Channel 2");  
        NewsChannel channel3 = new NewsChannel("Channel 3");  
  
        agency.attach(channel1);  
        agency.attach(channel2);  
        agency.attach(channel3);  
  
        // 뉴스 업데이트 - 모든 observer들이 자동으로 알림 받음  
        agency.setNews("Breaking News: Observer Pattern Example!");  
        // 출력:  
        // Channel 1 received news: Breaking News: Observer Pattern Example!  
        // Channel 2 received news: Breaking News: Observer Pattern Example!  
        // Channel 3 received news: Breaking News: Observer Pattern Example!  
  
        // Observer 제거  
        agency.detach(channel2);  
  
        // 새로운 뉴스 - channel2는 알림을 받지 않음  
        agency.setNews("Update: Channel 2 unsubscribed");  
        // 출력:  
        // Channel 1 received news: Update: Channel 2 unsubscribed  
        // Channel 3 received news: Update: Channel 2 unsubscribed  
    }  
}
```

## 장점

- 느슨한 결합: Subject와 Observer가 인터페이스를 통해 상호작용
- 동적 관계: 런타임에 observer를 추가/제거 가능
- 브로드캐스트 통신: 한 번의 변경으로 여러 객체에 알림
- 개방-폐쇄 원칙: 기존 코드 수정 없이 새로운 observer 추가 가능
- 재사용성: Subject와 Observer를 독립적으로 재사용 가능

## 단점

- 메모리 누수 위험: Observer 등록 후 해제하지 않으면 메모리 누수 발생 가능 (Lapsed Listener Problem)
- 예측 불가능한 업데이트 순서: Observer들의 알림 순서가 보장되지 않음
- 의도치 않은 업데이트: 작은 변경에도 모든 observer가 알림을 받을 수 있음
- 성능 문제: Observer가 많을 경우 알림 오버헤드 발생
- 디버깅 어려움: 간접적인 관계로 인해 흐름 추적이 어려울 수 있음

## Observer vs Publish-Subscribe

| 측면     | Observer Pattern | Publish-Subscribe Pattern |
|--------|------------------|---------------------------|
| 결합도    | 더 강한 결합 (직접 참조)  | 느슨한 결합 (중개자 사용)           |
| 통신 방식  | 직접 통신            | 간접 통신 (메시지 브로커)           |
| 동기/비동기 | 주로 동기적           | 주로 비동기적                   |
| 확장성    | 제한적              | 높은 확장성                    |
| 필터링    | 제한적              | 토픽 기반 필터링 가능              |
| 사용 예시  | GUI 이벤트, MVC     | 메시징 시스템, 마이크로서비스          |

# 메모리 누수 문제와 해결책

Observer가 등록된 후 해제되지 않으면 메모리 누수가 발생할 수 있습니다.

## 해결 방법

1. 명시적 해제: Observer를 사용 후 반드시 detach() 호출
2. 약한 참조(Weak Reference) 사용:

```
Iterator<WeakReference<Observer>> iterator = observers.iterator();
while (iterator.hasNext()) {
    Observer observer = iterator.next().get();
    if (observer == null) {
        iterator.remove(); // 가비지 컬렉션된 observer 제거
    } else {
        observer.update();
    }
}
```

# 성능 최적화

## 1. 스로틀링(Throttling)





빈번한 업데이트를 제한:

```
class ThrottledSubject {  
    private long lastNotifyTime = 0;  
    private static final long THROTTLE_PERIOD = 1000; // 1초  
  
    public void notifyObservers() {  
        long currentTime = System.currentTimeMillis();  
        if (currentTime - lastNotifyTime > THROTTLE_PERIOD) {  
            // 실제 알림 전송  
            lastNotifyTime = currentTime;  
        }  
    }  
}
```







# 요약

옵저버 패턴은 다음과 같은 경우에 유용함:

-  한 객체의 변경이 다른 객체들에게 전파되어야 할 때
-  이벤트 기반 시스템을 구축할 때
-  객체 간 느슨한 결합이 필요할 때
-  런타임에 관계를 동적으로 변경해야 할 때

옵저버 패턴은 다음과 같은 경우에 피해함:

-  Observer가 매우 많아 성능 문제가 예상될 때
-  알림 순서가 중요할 때
-  단순한 일대일 관계만 필요할 때
-  복잡한 필터링이나 라우팅이 필요할 때 (Pub-Sub 사용 권장)

감사합니다.

[aiiiiiiden@gmail.com](mailto:aiiiiiiden@gmail.com)