

**dart/flutter에 Observer 패턴 적용하기**

# 1. Observer 패턴에 효과적인 유즈케이스

옵저버 패턴은 다음과 같은 특징을 가진 유즈케이스에서 매우 효과적

- 일대다(one-to-many) 의존 관계가 필요한 경우
  - 하나의 상태 변경이 여러 객체에 영향을 미침
  - 각 Observer가 독립적으로 반응
- 느슨한 결합(loose coupling)이 필요한 경우
  - Subject와 Observer가 서로의 구체적인 구현을 몰라도 됨
  - 런타임에 Observer를 동적으로 추가/제거 가능
- 이벤트 기반 시스템을 구축하는 경우
  - UI 컴포넌트 간 상태 동기화
  - 실시간 데이터 업데이트

## 2. Counter 유즈케이스가 효과적인 이유

- 명확한 상태 변경 🖱️ 카운터 증가/감소/리셋이라는 단순하고 명확한 상태 변화
- 다양한 표현 방식 🖱️ 텍스트, 차트, 프로그레스 바, 색상 등 여러 형태로 동일한 데이터 표시
- 실시간 동기화 🖱️ 모든 Observer가 즉시 업데이트
- 독립적인 Observer 🖱️ 각 Observer가 서로 영향을 주지 않음
- 확장성 🖱️ 새로운 Observer 추가가 쉬움

### 3. Counter 앱의 비즈니스 요구사항

### 3.1. Primary Counter (메인 카운터)

- 큰 글씨로 현재 카운터 값 표시
- 파란색으로 강조
- 가장 눈에 띄는 위치에 배치

## 3.2. Secondary Counter (보조 카운터)

- 중간 크기 글씨로 동일한 값 표시
- 보라색으로 표시
- 메인 카운터와 동기화

### 3.3. Progress Bar (진행 바)

- 0부터 최대값(20)까지의 진행도를 시각적으로 표시
- 현재 값 / 최대값 형태로 표시
- 최대값 초과 시 100%로 표시

### 3.4. Color Box (색상 박스)

- 카운터 값에 따라 7가지 색상 중 하나로 변경
- 박스 안에 현재 값 표시
- 부드러운 색상 전환 애니메이션



### 3.5. Mood Icon (감정 아이콘)

- 카운터 값에 따라 다른 감정 아이콘 표시
- 음수: 빨간색, 0: 회색, 양수: 오렌지/파란색/초록색
- 7가지 아이콘 중 순환 표시

### 3.6. History Chart (히스토리 차트)

- 최근 20개의 카운터 값을 막대 그래프로 표시
- 양수: 파란색, 음수: 빨간색
- 최소값~최대값 범위 표시

## 4. Counter 앱의 기술적 요구사항

### 1. 자동 알림

☞ 카운터 값이 변경되면 모든 Observer에게 자동으로 알림

### 2. 느슨한 결합

☞ Subject는 Observer의 구체적인 타입을 알 필요 없음

### 3. 메모리 관리

☞ Widget dispose 시 Observer 등록 해제로 메모리 누수 방지

### 4. 테스트 가능성

☞ Subject와 Observer를 독립적으로 테스트 가능

## 5. 단위테스트로 요구사항 리뷰

## 5.1. CounterSubject 테스트 케이스 : 초기 상태 검증

초기 카운터 값은 0

```
test('초기 카운터 값은 0이어야 한다', () {  
  expect(subject.count, 0);  
});
```

## 5.2. CounterSubject 테스트 케이스 : 상태 변경 검증

increment, decrement, reset 동작 확인

```
test('increment() 호출 시 카운터가 1 증가해야 한다', () {  
  subject.increment();  
  expect(subject.count, 1);  
  
  subject.increment();  
  expect(subject.count, 2);  
});  
  
test('reset() 호출 시 카운터가 0으로 초기화되어야 한다', () {  
  subject.increment();  
  subject.increment();  
  expect(subject.count, 3);  
  
  subject.reset();  
  expect(subject.count, 0);  
});
```

## 5.3. Observer 등록/해제 테스트 케이스

Observer를 안전하게 등록하고 해제

```
test('Observer를 등록할 수 있어야 한다', () {  
  expect(() => subject.attach(observer1), returnsNormally);  
  expect(() => subject.attach(observer2), returnsNormally);  
});
```

```
test('Observer를 해제할 수 있어야 한다', () {  
  subject.attach(observer1);  
  expect(() => subject.detach(observer1), returnsNormally);  
});
```

## 5.4 알림 기능 테스트 케이스

모든 Observer에게 알림 전송

```
test('increment 시 모든 등록된 Observer에게 알림을 보내야 한다', () {  
    subject.attach(observer1);  
    subject.attach(observer2);  
    subject.attach(observer3);  
  
    subject.increment();  
  
    expect(observer1.updateCallCount, 1);  
    expect(observer2.updateCallCount, 1);  
    expect(observer3.updateCallCount, 1);  
});
```



## 5.5. 알림 내용 검증 테스트 케이스

Observer에게 올바른 값이 전달됨

```
test('Observer에게 올바른 값이 전달되어야 한다', () {  
    subject.attach(observer1);  
  
    subject.increment(); // 1  
    expect(observer1.lastReceivedValue, 1);  
  
    subject.increment(); // 2  
    expect(observer1.lastReceivedValue, 2);  
  
    subject.decrement(); // 1  
    expect(observer1.lastReceivedValue, 1);  
  
    subject.reset(); // 0  
    expect(observer1.lastReceivedValue, 0);  
});
```

## 5.6. Observer 해제 후 알림 테스트

해제된 Observer는 알림을 받지 않음

```
test('해제된 Observer는 알림을 받지 않아야 한다', () {  
  subject.attach(observer1);  
  subject.attach(observer2);  
  
  subject.increment();  
  expect(observer1.updateCallCount, 1);  
  expect(observer2.updateCallCount, 1);  
  
  subject.detach(observer1);  
  observer1.reset();  
  
  subject.increment();  
  expect(observer1.updateCallCount, 0); // 해제되어 알림 받지 않음  
  expect(observer2.updateCallCount, 2); // 계속 알림 받음  
});
```

## 5.7. 복잡한 시나리오 테스트

여러 Observer의 동적 등록/해제

```
test('여러 Observer가 서로 다른 시점에 등록/해제되어도 정상 동작해야 한다', () {  
  subject.attach(observer1);  
  subject.increment(); // observer1만 알림  
  
  subject.attach(observer2);  
  subject.increment(); // observer1, observer2 알림  
  
  subject.attach(observer3);  
  subject.increment(); // observer1, observer2, observer3 알림  
  
  subject.detach(observer2);  
  subject.increment(); // observer1, observer3 알림  
  
  expect(observer1.updateCallCount, 4);  
  expect(observer2.updateCallCount, 2);  
  expect(observer3.updateCallCount, 2);  
});
```

## 6. 코드 리뷰

## 6.1. Observer 인터페이스 정의

lib/observer/observer.dart

```
/// Observer 인터페이스
/// Subject의 상태 변경을 감지하기 위한 기본 인터페이스
abstract class Observer {
  /// Subject의 상태가 변경되었을 때 호출되는 메서드
  void update(int value);
}
```

## 6.2. Subject 인터페이스 정의

lib/subject/subject.dart

```
/// Subject 인터페이스
/// Observer들을 관리하고 알림을 보내는 기본 인터페이스
abstract class Subject {
  /// Observer를 등록
  void attach(Observer observer);

  /// Observer를 해제
  void detach(Observer observer);

  /// 모든 Observer들에게 알림
  void notifyObservers();
}
```

## 6.3. Observer/Subject 인터페이스의 역할

역할:

- 모든 Observer와 Subject의 공통 계약(contract) 정의
- 느슨한 결합(loose coupling) 구현
- 다형성(polymorphism)을 통한 유연한 설계

특징:

- 추상 클래스로 정의 (Dart의 `abstract class` )
- Subject는 Observer의 구체적인 타입을 몰라도 됨
- 새로운 Observer 추가 시 기존 코드 수정 불필요

## 6.4. CounterSubject - ConcreteSubject 구현





lib/subject/counter\_subject.dart

```
class CounterSubject implements Subject {  
  final List<Observer> _observers = [];  
  int _count = 0;  
  
  int get count => _count;  
  
  @override  
  void attach(Observer observer) {  
    _observers.add(observer);  
    print('Observer 등록됨: ${observer.runtimeType}');  
  }  
  
  @override  
  void detach(Observer observer) {  
    _observers.remove(observer);  
    print('Observer 해제됨: ${observer.runtimeType}');  
  }  
  
  @override  
  void notifyObservers() {  
    print('모든 Observer들에게 알림 전송 (현재 값: $_count)');  
    for (var observer in _observers) {  
      observer.update(_count);  
    }  
  }  
  
  void increment() {  
    _count++;  
    notifyObservers();  
  }  
}
```



## 6.5. CounterSubject의 특징

구현 포인트:

-  Observer 목록을 List로 관리
-  상태 변경 시 자동으로 `notifyObservers()` 호출
-  각 메서드에 로그 출력으로 동작 확인 가능
-  Observer의 구체적인 타입을 알 필요 없음

주의사항:

- Observer 목록은 `private ( _observers )`
- 상태도 `private ( _count )`로 캡슐화
- `public getter`로 읽기만 허용

## 6.6. TextObserver - ConcreteObserver 구현

lib/concrete\_observers/text\_observer.dart

```
class TextObserverState extends State<TextObserver> implements Observer {
  int _currentValue = 0;

  @override
  void update(int value) {
    setState(() {
      _currentValue = value;
    });
    print('${widget.label} 업데이트: $_currentValue');
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text(widget.label, style: TextStyle(fontSize: 14, fontWeight: FontWeight.bold)),
        SizedBox(height: 4),
        Text('$_currentValue', style: widget.textStyle ?? TextStyle(fontSize: 32, fontWeight: FontWeight.bold)),
      ],
    );
  }
}
```

## 6.7. Flutter에서 Observer 패턴 구현 시 주의사항

### StatefulWidget과의 통합:

- State 클래스가 Observer 인터페이스를 구현
- `update()` 메서드에서 `setState()` 호출하여 UI 갱신
- `GlobalKey`를 사용하여 State 인스턴스에 접근

### 메모리 관리:

- `dispose()` 에서 반드시 Observer 해제
- 메모리 누수 방지 (Lapsed Listener Problem)

## 6.8. Observer 구현체 간 비교

Observer	특징	구현 방식
TextObserver	텍스트로 값 표시	setState()로 UI 갱신
ProgressBarObserver	진행 바로 표시	최대값 대비 비율 계산
ColorBoxObserver	색상 변경	값에 따라 7가지 색상 순환
IconObserver	아이콘 변경	값에 따라 감정 아이콘 표시
ChartObserver	히스토리 차트	리스트로 이력 관리

공통점: 모두 Observer 인터페이스 구현, update() 메서드로 알림 수신

## 7. Observer 패턴의 장점

- 느슨한 결합 🖱️ Subject와 Observer가 인터페이스로만 연결
- 개방-폐쇄 원칙 🖱️ 새로운 Observer 추가 시 기존 코드 수정 불필요
- 동적 관계 🖱️ 런타임에 Observer 등록/해제 가능
- 브로드캐스트 🖱️ 한 번의 변경으로 여러 객체에 알림
- 재사용성 🖱️ Subject와 Observer를 독립적으로 재사용

## 8. Observer 패턴의 단점

- 메모리 누수 위험 🖱️ Observer 해제를 잊으면 메모리 누수 발생
- 예측 불가능한 순서 🖱️ Observer 알림 순서가 보장되지 않음
- 성능 문제 🖱️ Observer가 많으면 알림 오버헤드 증가
- 디버깅 어려움 🖱️ 간접적인 관계로 흐름 추적이 어려움
- 구현 복잡도 🖱️ ValueNotifier보다 코드가 복잡함

## 9. Observer 패턴 vs Pub-Sub 패턴

측면	Observer Pattern	Publish-Subscribe
결합도	Subject-Observer 직접 참조	중개자(Event Bus) 사용
통신	동기적, 직접 호출	비동기적, 큐 기반
필터링	제한적	토픽/채널 기반 필터링
확장성	제한적	높음
사용 예시	UI 상태 관리	마이크로서비스, 메시징

## 10. Observer 패턴 vs ValueNotifier

측면	Observer Pattern	ValueNotifier
구조	Subject/Observer 인터페이스	ChangeNotifier 상속
데이터 전달	Push 모델 (update에 값 전달)	Pull 모델 (value 접근)
타입 안전성	Observer가 타입 명시	리스너가 타입 모름
복잡도	높음 (인터페이스, 등록 관리)	낮음 (내장 기능)
유연성	높음 (다양한 데이터 전달 가능)	제한적 (단일 값만)
학습 목적	✅ 패턴 이해에 적합	❌ 추상화되어 내부 구조 파악 어려움
실무 사용	특수한 경우	일반적인 상태 관리



## 10.1. ValueNotifier가 엄밀한 Observer 패턴이 아닌 이유

### 1. Pull 모델 사용

```
// 전통적 Observer Pattern (Push)
interface Observer {
    void update(int newValue); // 데이터를 받음
}





// ValueNotifier (Pull)
counter.addListener(() {
    print(counter.value); // 데이터를 가져와야 함
});
```

### 2. Observer 인터페이스 부재

- 명시적인 Observer 인터페이스가 없음
- 단순 콜백(VoidCallback) 사용
- 어떤 notifier인지 context 없음

## 10.2. ValueNotifier의 장점

장점:

-  Flutter 내장 기능으로 간단함
-  ValueListenableBuilder로 UI 통합 쉬움
-  보일러플레이트 코드 최소화
-  dispose 자동 관리 (ValueListenableBuilder 사용 시)

## 10.3 ValueNotifier를 사용한 구현 예시

lib/observer\_pattern\_demo2.dart

```
class _ObserverPatternDemo2State extends State<ObserverPatternDemo2> {
  final ValueNotifier<int> _counterNotifier = ValueNotifier<int>(0);

  @override
  Widget build(BuildContext context) {
    return ValueListenableBuilder<int>(
      valueListenable: _counterNotifier,
      builder: (context, value, child) {
        return Text('$value', style: TextStyle(fontSize: 32));
      },
    );
  }

  void _increment() {
    _counterNotifier.value++; // 모든 리스너에게 자동 알림
  }
}
```

## 11. Flutter의 다른 상태 관리와 비교

방식	Observer 패턴 여부	특징
Custom Observer Pattern	✅ 엄밀한 패턴	학습용, 완전한 제어
ValueNotifier	△ Listener 패턴	간단한 상태 관리
ChangeNotifier	△ Listener 패턴	Provider와 함께 사용
Stream	✅ 비동기 Observer	RxDart, BLoC
Provider	△ InheritedWidget	의존성 주입 + 상태 관리
Riverpod	△ 선언적 상태 관리	Provider 개선 버전

감사합니다.

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