

**dart/flutter에 Strategy 패턴 적용하기**

# 1. Strategy 패턴에 효과적인 유즈케이스

전략 패턴은 다음과 같은 특징을 가진 유즈케이스에서 매우 효과적

- 런타임에 알고리즘을 선택해야 하는 경우
  - 사용자 선택이나 환경에 따라 동작이 달라짐
  - 같은 작업을 여러 방식으로 수행 가능
- 관련된 알고리즘군이 존재하는 경우
  - 동일한 인터페이스를 공유하지만 구현이 다르며, 알고리즘 간 독립적으로 변경 가능
- 조건문 대신 다형성을 사용하고 싶은 경우
  - if-else나 switch 문을 클래스로 대체
  - 개방-폐쇄 원칙(OCP) 준수

## 2. 데이터 포맷 변환 유즈케이스가 효과적인 이유

- 다양한 포맷 지원 🖱️ JSON, CSV, XML 등 여러 포맷으로 데이터 변환
- 런타임 선택 🖱️ 사용자가 UI에서 원하는 포맷 선택
- 독립적인 알고리즘 🖱️ 각 포맷의 직렬화 로직이 서로 독립적
- 확장성 🖱️ 새로운 포맷 추가가 쉬움 (YAML, TOML 등)
- 일관성 🖱️ 모든 serializer가 동일한 인터페이스 사용

### 3. 데이터 포맷 변환의 비즈니스 요구사항

## 3.1. JSON (JavaScript Object Notation)

- 웹 API의 표준 데이터 포맷
- 계층적 데이터 구조 표현
- JavaScript와 호환성 우수
- MIME 타입: `application/json`

## 3.2. CSV (Comma-Separated Values)

- 스프레드시트 애플리케이션과 호환
- 단순한 표 형태의 데이터 표현
- 파일 크기가 작고 처리 속도 빠름
- MIME 타입: `text/csv`

### 3.3. XML (eXtensible Markup Language)

- 레거시 시스템과 호환성
- 스키마 정의 가능 (XSD)
- 복잡한 계층 구조 표현
- MIME 타입: `application/xml`

## 4. 데이터 포맷 변환의 기술적 요구사항

### 1. 양방향 변환 지원

- ☞ 직렬화(serialize)와 역직렬화(deserialize) 모두 지원

### 2. 타입 안전성

- ☞ 강타입 언어의 장점을 활용하여 컴파일 타임 체크

### 3. 에러 처리

- ☞ 잘못된 포맷 데이터에 대한 명확한 예외 처리

### 4. 확장 가능한 구조

- ☞ 새로운 포맷 추가 시 기존 코드 수정 최소화



## 5. 단위테스트로 요구사항 리뷰

## 5.1. JsonSerializer 테스트 케이스 : 직렬화 검증

올바른 JSON 배열 형태로 변환, 필드 이름과 값 포함

```
test('should serialize single user to JSON', () {  
  const users = [  
    User(  
      id: '1',  
      name: 'Alice',  
      email: 'alice@example.com',  
      age: 28,  
      role: 'Developer',  
    ),  
  ];  
  
  final result = serializer.serialize(users);  
  
  expect(result, contains('"id":"1"'));  
  expect(result, contains('"name":"Alice"'));  
  expect(result, contains('"email":"alice@example.com"'));  
});
```

## 5.2. JsonSerializer 테스트 케이스 : 역직렬화 검증

JSON 문자열을 User 객체 리스트로 변환

```
test('should deserialize JSON to users list', () {  
    const jsonString = '[{"id":"1","name":"Alice","email":"alice@example.com","age":28,"role":"Developer"}]';  
  
    final users = serializer.deserialize(jsonString);  
  
    expect(users.length, 1);  
    expect(users[0].id, '1');  
    expect(users[0].name, 'Alice');  
});
```

## 5.3. JsonSerializer 테스트 케이스 : Round-trip 검증

직렬화 후 역직렬화했을 때 원본 데이터 유지

```
test('should perform round-trip serialization', () {  
  const originalUsers = [  
    User(id: '1', name: 'Alice', email: 'alice@example.com', age: 28, role: 'Developer'),  
  ];  
  
  final serialized = serializer.serialize(originalUsers);  
  final deserialized = serializer.deserialize(serialized);  
  
  expect(deserialized.length, originalUsers.length);  
  expect(deserialized[0].id, originalUsers[0].id);  
  expect(deserialized[0].name, originalUsers[0].name);  
});
```

## 5.4 CsvSerializer 테스트 케이스 : 헤더 포함 검증

CSV는 첫 행에 컬럼 헤더 포함

```
test('should serialize with CSV headers', () {  
  const users = [User(...)];  
  
  final result = serializer.serialize(users);  
  
  expect(result, contains('id,name,email,age,role'));  
});
```

## 5.5. CsvSerializer 테스트 케이스 : 특수문자 처리

쉼표 포함 데이터는 따옴표로 감싸기

```
test('should handle commas in user data', () {  
  const users = [  
    User(id: '1', name: 'Smith, John', email: 'john@example.com', age: 30, role: 'Developer'),  
  ];  
  
  final result = serializer.serialize(users);  
  
  expect(result, contains('"Smith, John"'));  
});
```

## 5.6. XmlSerializer 테스트 케이스 : XML 구조 검증

올바른 XML 선언과 계층 구조

```
test('should serialize to valid XML structure', () {  
    const users = [User(...)];  
  
    final result = serializer.serialize(users);  
  
    expect(result, contains('<?xml version="1.0" encoding="UTF-8"?>'));  
    expect(result, contains('<users>'));  
    expect(result, contains('<user>'));  
    expect(result, contains('</user>'));  
    expect(result, contains('</users>'));  
});
```

## 5.7. XmlSerializer 테스트 케이스 : 특수문자 이스케이핑

XML 특수문자를 올바르게 이스케이프

```
test('should handle special XML characters', () {  
  const users = [  
    User(id: '1', name: 'Alice & Bob', email: 'test@example.com', age: 30, role: 'Developer <Senior>'),  
  ];  
  
  final result = serializer.serialize(users);  
  
  expect(result, contains('Alice & Bob'));  
  expect(result, contains('Developer &lt;Senior>'));  
});
```



## 6. 코드 리뷰



## 6.1. User 모델 클래스

lib/models/user.dart

```
class User {
  final String id;
  final String name;
  final String email;
  final int age;
  final String role;

  const User({required this.id, required this.name, required this.email, required this.age, required this.role});

  /// JSON 직렬화
  Map<String, dynamic> toJson() {
    return {'id': id, 'name': name, 'email': email, 'age': age, 'role': role};
  }

  /// JSON 역직렬화
  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      id: json['id'] as String,
      name: json['name'] as String,
      email: json['email'] as String,
      age: json['age'] as int,
      role: json['role'] as String,
    );
  }
}
```

## 6.3. Strategy 인터페이스의 역할과 특징

역할:

- 모든 serializer의 공통 계약(contract) 정의
- 클라이언트 코드가 구체적인 구현에 의존하지 않도록 함
- 런타임에 전략을 교체 가능하게 만듦

특징:

- 추상 클래스로 정의 (Dart의 `abstract class` )
- 모든 메서드가 추상 메서드
- 구현 클래스는 모든 메서드를 반드시 구현해야 함

## 6.4. JsonSerializer - JSON 전략 구현

lib/strategies/json\_serializer.dart





```
class JsonSerializer implements DataSerializerStrategy {
  @override
  String serialize(List<User> users) {
    final List<Map<String, dynamic>> jsonList = users.map((user) => user.toJson()).toList();
    return jsonEncode(jsonList);
  }
  @override
  List<User> deserialize(String data) {
    final List<dynamic> jsonList = jsonDecode(data) as List<dynamic>;

    return jsonList
      .map((json) => User.fromJson(json as Map<String, dynamic>))
      .toList();
  }
  @override
  String getFormatName() => 'JSON';
  @override
  String getFileExtension() => '.json';
  @override
  String getMimeType() => 'application/json';
}
```

## 6.5. CsvSerializer - CSV 전략 구현

lib/strategies/csv\_serializer.dart

구현 포인트:



-  csv 패키지 사용 ( ListToCsvConverter , CsvToListConverter )
-  헤더 행 자동 생성 및 처리
-  심표 포함 데이터는 자동으로 따옴표 처리
-  타입 변환 처리 (age를 int로 파싱)

```
@override
String serialize(List<User> users) {
  final List<List<dynamic>> csvData = [
    ['id', 'name', 'email', 'age', 'role'], // 헤더
    ...users.map((user) => [user.id, user.name, user.email, user.age, user.role,]),
  ];
  return const ListToCsvConverter().convert(csvData);
}
```

## 6.6. XmlSerializer - XML 전략 구현

lib/strategies/xml\_serializer.dart

구현 포인트:

-  xml 패키지 사용 ( XmlBuilder , XmlDocument )
-  특수문자 자동 이스케이핑 ( & , < , > )

```
@override String serialize(List<User> users) {  
    final builder = XmlBuilder();  
    builder.processing('xml', 'version="1.0" encoding="UTF-8"');  
    builder.element('users', nest: () {  
        for (final user in users) {  
            builder.element('user', nest: () { /* ... 나머지 필드 */ });  
        }  
    });  
    return builder.buildDocument().toXmlString(pretty: true);  
}
```

## 6.7. DataConverter - Context 클래스

lib/context/data\_converter.dart

```
/// Context 클래스 - Strategy를 사용하는 클라이언트
class DataConverter {
  DataSerializerStrategy? _strategy;
  /// 전략 설정 (런타임에 변경 가능)
  void setStrategy(DataSerializerStrategy strategy) => _strategy = strategy;
  /// 현재 전략을 사용하여 직렬화
  String serialize(List<User> users) => _strategy!.serialize(users);
  /// 현재 전략을 사용하여 역직렬화
  List<User> deserialize(String data) => _strategy!.deserialize(data);
  /// 현재 포맷 이름 조회
  String? getFormatName() => _strategy?.getFormatName();
}
```



## 6.8. DataConverter의 역할과 특징

역할:

- Strategy 객체를 보유하고 사용
- 클라이언트로부터 구체적인 전략을 숨김
- 런타임에 전략 교체 지원

특징:

- Strategy에 대한 참조를 멤버 변수로 유지
- Setter를 통해 전략을 주입받음 (Dependency Injection)
- Null 체크로 전략 미설정 시 명확한 에러 제공

## 6.9. Flutter UI에서 Strategy 패턴 사용

lib/main.dart

```
class _DataConverterPageState extends State<DataConverterPage> {  
  // 중략...  
  void _changeFormat(String format) {  
    setState(() {  
      _selectedFormat = format;  
      switch (format) {  
        case 'JSON': _converter.setStrategy(JsonSerializer()); // 전략 교체  
        case 'CSV': _converter.setStrategy(CsvSerializer());  
        case 'XML': _converter.setStrategy(XmlSerializer());  
      }  
    });  
  }  
  void _convertData() {  
    final serialized = _converter.serialize(_sampleUsers);  
    setState(() => _output = serialized);  
  }  
}
```





## 6.10. Strategy 구현체 간 비교

Strategy	Package	특징	사용 케이스
JsonSerializer	dart:convert	계층 구조, 웹 표준	REST API, 웹 앱
CsvSerializer	csv	표 형태, 경량	엑셀 호환, 데이터 분석
XmlSerializer	xml	스키마 정의, 레거시	SOAP API, 설정 파일





## 7. Strategy 패턴 vs Template Method 패턴 비교

측면	Strategy	Template Method
관계	합성 (Composition)	상속 (Inheritance)
변경 시점	런타임	컴파일 타임
전체 알고리즘	Context가 결정	부모 클래스가 결정
변경 범위	전체 알고리즘 교체	일부 단계만 변경
확장성	새 전략 추가 쉬움	새 템플릿 추가는 상속 필요
복잡도	더 많은 클래스	더 적은 클래스

## Strategy 패턴을 사용하기 좋은 경우:

-  런타임에 알고리즘을 선택해야 할 때
-  알고리즘이 자주 변경되거나 추가될 때
-  복잡한 조건문을 제거하고 싶을 때
-  클라이언트가 구현 세부사항을 알 필요가 없을 때

## Template Method를 사용하기 좋은 경우:

-  알고리즘의 순서가 고정되어 있을 때
-  일부 단계만 커스터마이징이 필요할 때
-  공통 로직이 많고 변경되는 부분이 적을 때
-  상속 관계가 자연스러울 때

## 9. Strategy 패턴의 장점

- 개방-폐쇄 원칙 (OCP) ➡ 기존 코드 수정 없이 새로운 전략 추가 가능
- 단일 책임 원칙 (SRP) ➡ 각 전략이 하나의 책임만 가짐
- 런타임 유연성 ➡ 실행 중에 알고리즘을 동적으로 변경 가능
- 테스트 용이성 ➡ 각 전략을 독립적으로 테스트 가능
- 조건문 제거 ➡ if-else, switch 대신 다형성 사용

## 10. Strategy 패턴의 단점

- 클래스 증가 🖱 전략마다 별도의 클래스 필요하며, 코드의 복잡도 증가
- 클라이언트의 인지 필요 🖱 전략 간 차이를 이해하고, 클라이언트가 적절한 전략을 선택해야 함
- 오버엔지니어링 위험 🖱 전략이 1-2개뿐이면 과도함. 단순한 경우 조건문이 더 나옴

감사합니다.

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