**디자인 패턴**

**디자인 패턴의 정의**

* ﻿﻿디자인 패턴은 소프트웨어 개발 과정에서 자주 발생하는 문제들에 대한 전문가들의 검증된 해결책입니다.
* ﻿﻿이는 수년간의 소프트웨어 개발 경험을 통해 발견된 best practices를 체계화한 것입니다.
* ﻿﻿특정 상황에서 공통적으로 발생하는 문제에 대한 재사용 가능한 해결책을 제공합니다.
* ﻿﻿구체적인 구현이 아닌 문제 해결을 위한 템플릿으로, 다양한 상황에 맞게 응용할 수 있습니다.

﻿﻿객체 지향 설계 원칙을 따르며, 코드의 재사용성, 유지보수성, 확장성을 높이는 데 중점을 둡니다.

**디자인 패턴의 중요성**

* ﻿﻿**개발 시간 단축**: 검증된 해결책을 사용함으로써 문제 해결에 소요되는 시간을 크게 줄일 수 있습니다.
* ﻿﻿**코드 품질 향상**: 구조화된 방식으로 설계하여 코드의 가독성, 유지보수성, 신뢰성을 높입니다.
* ﻿﻿**확장성 확보:** 미래의 요구사항 변경에 유연하게 대응할 수 있는 시스템 구조를 설계할 수 있습니다.
* ﻿﻿**의사소통 개선**: 개발자 간에 공통된 어휘를 제공하여 설계 아이디어를 효과적으로 전달할 수 있습니다.
* ﻿﻿**재사용성 증대**: 검증된 솔루션을 여러 프로젝트에 걸쳐 재사용함으로써 생산성을 높일 수 있습니다.
* ﻿﻿**복잡성 관리:** 대규모 시스템을 체계적으로 구조화하여 복잡성을 효과적으로 관리할 수 있습니다.

﻿﻿**모범 사례 적용:** 경험 많은 개발자들의 지식을 활용하여 높은 품질의 소프트웨어를 개발할 수 있습니다.

**MVC (Model-View-Controller) 패턴**

* ﻿﻿MVC는 애플리케이션을 세 가지 주요 컴포넌트로 분리하는 아키텍처 패턴입니다.
* ﻿﻿사용자 인터페이스와 비즈니스 로직을 분리하여 유지보수성과 확장성을 향상시킵니다.

구성 요소

1. ﻿﻿﻿Model: 데이터와 비즈니스 로직을 관리
2. ﻿﻿﻿View: 사용자에게 정보를 표시

﻿﻿﻿Controller: Model과 View 사이의 상호작용을 조정

**프로세스 흐름**

* ﻿﻿﻿사용자가 View를 통해 백엔드 서버로 요청을 보냄
* ﻿﻿﻿서버의 Controller가 요청을 받아 처리
* ﻿﻿﻿Controller가 필요시 Model을 업데이트
* ﻿﻿﻿Model이 변경되면 View에 반영
* ﻿﻿﻿업데이트된 View가 사용자에게 표시

**장점**

1. ﻿﻿관심사의 분리로 코드 유지보수 용이
2. ﻿﻿각 컴포넌트의 독립적인 개발 가능
3. ﻿﻿테스트 용이성 증가

**단점**

* ﻿﻿복잡한 애플리케이션에서는 컨트롤러가 비대해질 수 있음

﻿﻿세 가지 주요 컴포넌트의 역할과 상호작용을 명확히 이해해야 함

**MVC 변형 패턴**

* **﻿﻿MVP (Model-View-Presenter)**
* ﻿﻿개요
* ﻿﻿MVC의 Controller를 Presenter로 대체한 패턴
* ﻿﻿View와 Model의 완전한 분리가 특징
* ﻿﻿구성
* ﻿﻿Model: 데이터와 비즈니스 로직 관리
* ﻿﻿View: 사용자 인터페이스, Presenter와 1:1 관계
* ﻿﻿Presenter: View와 Model 사이의 중재자 역할
* ﻿﻿특징
* ﻿﻿View와 Model 간 직접 통신 없음
* ﻿﻿Presenter가 View와 Model 모두 참조

﻿﻿테스트 용이성 증가

**MVC 변형 패턴**

* **﻿﻿MVVM (Model-View-ViewModel)**
* ﻿﻿개요
* ﻿﻿주로 비 개발에 사용되는 패턴
* ﻿﻿데이터 바인딩을 활용한 View와 ViewModel의 느슨한 결합이 특징
* ﻿﻿구성
* ﻿﻿Model: 데이터와 비즈니스 로직
* ﻿﻿View: 사용자 인터페이스
* ﻿﻿ViewModel: View를 위한 Model 표현, View와 Model 사이의 중재자
* ﻿﻿특징
* ﻿﻿데이터 바인딩으로 View와 ViewModel 간 자동 동기화
* ﻿﻿View의 상태와 동작을 ViewModel에서 관리

﻿﻿UI 플랫폼 독립적인 개발 가능

**MVC vs MVP vs MVVM**

* ﻿﻿데이터 흐름
* ﻿﻿MVC: Controller → Model → View
* ﻿﻿MVP: Model <> Presenter <> View
* ﻿﻿MVVM: Model ViewModel <> View (양방향 데이터 바인딩)
* ﻿﻿View와 Model의 결합도
* ﻿﻿MVC: 중간
* ﻿﻿MVP: 낮음
* ﻿﻿MVVM: 매우 낮음
* ﻿﻿사용 상황
* ﻿﻿MVC: 일반적인 웹 애플리케이션
* ﻿﻿MVP: 복잡한 UI 로직을 가진 애플리케이션

﻿﻿MVVM: 데스크톱 또는 모바일 애플리케이션, SPA

**싱글톤 (Singleton) 패턴**

* ﻿﻿싱글톤은 클래스의 인스턴스가 오직 하나만 생성되도록 보장하는 디자인 패턴입니다.
* ﻿﻿전역 접근 지점을 통해 어디서든지 인스턴스에 접근할 수 있도록 합니다.
* ﻿﻿리소스 공유, 상태 관리 등에 주로 사용됩니다.

**구현 방법**

1. ﻿﻿﻿생성자를 private으로 선언
2. ﻿﻿﻿클래스 내부에 자신의 정적 인스턴스를 보관
3. ﻿﻿﻿인스턴스에 접근할 수 있는 정적 메소드 제공

클래스 생성 없이 사용할 수 있는 메소드를 통해 모든 객체에서 공유할 수 있는 인스턴스를 조회

**싱글톤 (Singleton) 패턴**

* **﻿﻿장점**
* ﻿﻿**단일 인스턴스 보장:** 클래스의 인스턴스가 하나만 생성되므로, 메모리와 리소스를 절약할 수 있습니다.
* ﻿﻿**전역적 접근:** 어디서든지 인스턴스에 접근할 수 있는 전역적인 접근점을 제공하여 객체의 공유와 일관성을 유지할 수 있습니다.
* ﻿﻿**지연된 초기화**: 필요한 시점에서만 인스턴스를 생성하므로 초기화 과정이 지연되어 성능을 향상시킬 수 있습니다.
* ﻿﻿**단점**
* ﻿﻿**멀티스레드 환경에서 동기화 문제:** 동시에 여러 스레드에서 접근할 경우 인스턴스의 상태가 일관성 없이 될 수 있으므로 동기화 처리가 필요합니다.
* ﻿﻿**테스트 어려움:** 전역 상태를 유지하는 패턴이기 때문에 테스트 시 의존성 관리가 어려울 수 있습니다.

﻿﻿**단일 책임 원칙 위배 가능성:** 싱글톤 클래스가 너무 많은 책임을 맡게 되면 클래스의 책임이 모호해질 수 있습니다.

**싱글톤 (Singleton) 패턴**

* **﻿﻿사용 사례**
* ﻿﻿**커넥션 풀 관리**: 데이터베이스 연결과 같은 공유 자원을 단일 인스턴스로 관리할 때 유용합니다.

**• 설정 정보 관리:** 시스템 설정 정보나 로그 기록과 같은 공유 데이터를 처리할 때 사용할 수 있습니다.

1. ﻿﻿**캐시 관리:** 여러 곳에서 접근되는 캐시 객체를 싱글톤으로 관리하여 일관된 데이터 접근을 보장할 수 있습니다.
2. ﻿﻿**주의 사항**
3. **﻿﻿멀티스레드 환경에서의 동기화:** 멀티스레드 환경에서는 인스턴스의 동시 접근이 발생할 수 있으므로, 동기화 문제를 고려하여 구현해야 합니다.

﻿﻿**테스트 어려움:** 싱글톤은 전역 상태를 유지하므로 테스트할 때 의존성 관리가 어려울 수 있습니다.

**팩토리 패턴 (Factory Pattern)**

* ﻿﻿팩토리 패턴은 객체를 생성하는 인터페이스를 정의하고, 이를 통해 객체의 생성을 서브 클래스에게 위임하는 디자인 패턴입니다.
* ﻿﻿이 패턴을 사용하면 실제 생성되는 객체의 클래스를 몰라도 되며, 추상적인 인터페이스나 추상 클래스를 통해 객체를 생성하고 이를 반환받을 수 있습니다.

구성 요소

1. ﻿﻿**추상 팩토리 (Abstract Factory):** 객체 생성을 위한 인터페이스를 제공합니다. 팩토리 메서드 패턴과는 달리 객체의 구체적인 클래스를 지정하지 않습니다.
2. ﻿﻿**구체적 팩토리 (Concrete Factory):** 실제 객체를 생성하는 클래스입니다. 추상 팩토리를 구현하고 실제 객체를 생성하는 메서드를 구현합니다.

﻿﻿**제품 (Product):** 생성되는 객체를 나타냅니다. 팩토리는 이 제품을 생성하고 반환합니다.

**팩토리 패턴 (Factory Pattern)**

* **﻿﻿장점**
* **﻿﻿객체 생성의 중앙 집중화:** 객체 생성 로직을 한 곳에 모아 관리할 수 있습니다.
* ﻿﻿**유연성:** 클라이언트 코드는 구체적인 클래스를 몰라도 되므로, 객체 생성 로직의 변경이나 확장에 용이합니다.
* ﻿﻿**코드 중복 감소:** 동일한 객체 생성 코드를 여러 곳에 중복해서 사용하지 않고, 팩토리 메서드를 통해 중복을 제거할 수 있습니다.
* ﻿﻿**단점**
* **﻿﻿클래스 증가:** 객체의 종류가 많아질수록 구체적인 팩토리 클래스도 많아지며, 클래스의 수가 늘어날 수 있습니다.

﻿﻿**복잡성 증가:** 팩토리 패턴을 오버 엔지니어링할 경우 코드가 복잡해질 수 있습니다.

**옵저버 패턴 (Observer Pattern)**

* ﻿﻿옵저버 패턴은 객체 간의 일대다(one-to-many) 의존 관계를 정의하는 패턴으로, 어떤 객체의 상태가 변할 때 그 객체에 의존하는 다른 객체들이 자동으로 알림을 받고 자동으로 업데이트될 수 있도록 구성합니다.
* ﻿﻿이 패턴은 객체 간의 느슨한 결합을 가능하게 하며, 이벤트 기반 시스템에서 많이 활용됩니다.

구성 요소

1. ﻿﻿**Subject (주체):** 상태가 변경되는 주체 객체입니다. 일반적으로 상태를 관리하고, 옵저버들을 등록하고 알림을 보내는 메서드를 제공합니다.
2. ﻿﻿**Observer (옵저버):** 주체 객체의 상태 변화를 관찰하고 이에 반응하는 객체입니다. 주체에서 발생한 변화에 대한 알림을 받아 적절한 동작을 수행합니다.
3. ﻿﻿**ConcreteSubject (구체적 주체):** Subject 인터페이스를 구현하며, 옵저버들을 등록하고 상태 변화를 알리는 메서드를 구현합니다.

﻿﻿**ConcreteObserver (구체적 옵저버):** Observer 인터페이스를 구현하며, 주체 객체의 상태 변화에 따라 특정 동작을 수행하는 구체적인 클래스입니다.

**옵저버 패턴 (Observer Pattern)**

* **﻿﻿장점**
* **﻿﻿느슨한 결합 (Loose Coupling):** 주체 객체와 옵저버 객체 간의 관계가 느슨하여 상호 의존성이 줄어듭니다.
* ﻿﻿**변화 관리:** 상태 변화를 관리하고, 이에 따라 다수의 객체들이 자동으로 업데이트될 수 있습니다.
* ﻿﻿**확장성:** 새로운 옵저버를 추가하거나 기존 옵저버를 제거하는 것이 용이합니다.
* ﻿﻿**단점**
* **﻿﻿너무 많은 알림:** 옵저버가 많거나 업데이트 빈도가 높을 경우, 불필요한 알림이 발생할 수 있습니다.

﻿﻿**순서의 의존성:** 옵저버들의 처리 순서가 중요한 경우에는 따로 관리해주어야 합니다.

**전략 패턴 (Strategy Pattern)**

* ﻿﻿전략 패턴은 객체의 행위를 클래스로 캡슐화하고, 이 행위를 필요에 따라 교체할 수 있게 만드는 디자인 패턴입니다.
* ﻿﻿이 패턴은 특정한 알고리즘의 집합을 정의하고, 각각을 캡슐화하여 이들을 서로 교환할 수 있게 만듭니다.
* ﻿﻿이를 통해 클라이언트는 동일한 인터페이스를 통해 다양한 알고리즘을 적용할 수 있습니다.

**구성 요소**

1. **﻿﻿전략(Strategy):** 인터페이스나 추상 클래스를 통해 특정 행위(알고리즘) 를 정의합니다.
2. ﻿﻿**구체적인 전략(Concrete Strategy):** 전략 인터페이스를 구현한 실제 알고리즘 클래스들입니다.

﻿﻿**컨텍스트(Context):** 전략 객체를 사용하는 클라이언트입니다. 컨텍스트는 전략을 실행하는 메서드를 호출하여 해당 전략을 교체하거나 설정할 수 있습니다.

**전략 패턴 (Strategy Pattern)**

**!.**

* **﻿﻿장점**
* **﻿﻿유연성과 확장성:** 동일한 인터페이스를 사용하여 다양한 전략을 적용할 수 있습니다.
* ﻿﻿**코드 재사용:** 전략은 독립적으로 캡슐화되어 있어서, 다른 곳에서 재사용할 수 있습니다.
* ﻿﻿**알고리즘의 독립성:** 알고리즘이 컨텍스트와 분리되어 있어서, 알고리즘의 변경이 컨텍스트에 영향을 주지 않습니다.
* ﻿﻿**단점**
* **﻿﻿클라이언트가 전략을 명시적으로 선택해야 함:** 전략을 변경하기 위해서는 클라이언트 코드에서 명시적으로 새로운 전략을 설정해주어야 합니다.

﻿﻿**컨텍스트와 전략 객체 간의 복잡성:** 전략이 많고 복잡할 경우, 컨텍스트 클래스가 복잡해질 수 있습니다.

**의존성 주입 패턴 (Dependency Injection Pattern)**

* ﻿﻿의존성 주입 패턴은 객체 간의 의존 관계를 외부에서 설정해주는 디자인 패턴입니다.
* ﻿﻿이 패턴을 사용하면 객체는 직접 필요한 의존 객체를 생성하거나 결정하지 않고, 외부에서 주입받아 사용할 수 있습니다.
* ﻿﻿이는 객체 간의 결합도를 낮추고 유연성을 높이는 데 도움을 줍니다

주요 개념

1. ﻿﻿**의존성(Dependency):** 한 객체가 다른 객체를 사용할 때의 관계를 의미합니다. 예를 들어, 클래스 A가 클래스 B를 사용한다면 A는 B에 의존성이 있습니다.
2. ﻿﻿**의존성 주입(Dependency Injection):** 외부에서 객체의 의존성을 설정하거나 주입하는 것을 의미합니다. 주로 생성자 주입, setter 주입, 인터페이스 주입 등의 방법이 사용됩니다.

﻿﻿**주입 컨테이너(Injection Container):** 의존성을 관리하고 주입하는 역할을 수행하는 프레임워크나 라이브러리를 말합니다. 예를 들어, Spring Framework의 ApplicationContext가 주입 컨테이너의 역할을 합니다.

**의존성 주입 패턴 (Dependency Injection Pattern)**

* **﻿﻿장점**
* **﻿﻿유연성과 재사용성:** 객체 간의 결합도를 낮추어 재사용 가능한 코드를 작성할 수 있습니다.
* ﻿﻿**테스트 용이성:** 의존성을 모의(Mock) 객체로 쉽게 대체하여 테스트하기 용이합니다.
* ﻿﻿**관리 용이성:** 주입된 객체의 라이프사이클 관리를 주입 컨테이너에 위임할 수 있습니다.
* ﻿﻿**단점**
* **﻿﻿복잡성:** 설정이나 구성에 대한 추가적인 처리가 필요할 수 있습니다.

﻿﻿**초기 설정 오류:** 의존성 주입이 잘못 설정되었을 경우, 실행 시 예외가 발생할 수 있습니다.

**ORM (Object-Relational Mapping)**

* ﻿﻿ORM은 객체 지향 프로그래밍 언어와 관계형 데이터베이스 간의 데이터 변환을 자동화하는 기술입니다.
* ﻿﻿이 패턴은 객체를 데이터베이스의 테이블에 매핑하고, 객체 간의 관계를 유지하며 데이터 액세스를 추상화합니다.

**객체-관계 매핑 (Object-Relational Mapping)**

1. ﻿﻿객체 지향 프로그래밍에서는 객체 간의 관계를 코드로 쉽게 표현할 수 있지만, 관계형 데이터베이스는 테이블 형태로 데이터를 저장하고 관리합니다.
2. ﻿﻿ORM은 이 두 가지 사이의 불일치를 해결하기 위해 객체를 데이터베이스 테이블로 매핑하고, 객체 간의 관계를 테이블 간의 관계로 매핑합니다.

**ORM 도구**

• 대표적인 ORM 도구로는 Hibernate (Java), Django ORM (Python), Sequelize (Nodejs) 등이 있습니다.