1. **AuthenticationService - Singleton (생성 패턴)**

* Singleton은 인스턴스가 오직 하나만 생성되는 것을 보장하고, 어디서든 이 인스턴스에 접근할 수 있도록 하는 패턴이다.
* AuthenticationService 클래스는 시스템 전반에서 하나의 인증 시스템을 유지해야 하므로 Singleton이 적합하다. 이를 통해 로그인, 로그아웃과 같은 인증 메커니즘이 중복되지 않고 일관된 상태를 유지할 수 있다.

|  |
| --- |
| **AuthenticationService** |
| - instance: **AuthenticationService**  - authToken: String |
| - **AuthenticationService()**  + getInstance(): **AuthenticationService**  **+ login(String username, String password): Boolean**  + logout(String userID): void |

1. **ProductSearchEngine – Strategy (행동 패턴)**

* Strategy 패턴은 특정 알고리즘을 실행하는 부분을 인터페이스나 추상 클래스를 통해 정의하고, 실행할 알고리즘의 변형을 구현 클래스에서 제공하는 방식으로 작성된다. 이렇게 하면 알고리즘을 사용하는 클라이언트 코드에서는 구체적인 알고리즘 클래스를 바꿔 쓰면서도 변경 없이 동일한 방식으로 알고리즘을 사용할 수 있다.
* SearchStrategy는 추상 클래스로, 다양한 검색 알고리즘(CategorySearch, KeywordSearch)을 위한 인터페이스 역할을 한다. ProductSearchEngine 클래스는 getSearchStrategy 메소드를 통해 검색 유형에 맞는 구체적인 검색 전략 객체를 생성하고, 이 객체의 search 메소드를 호출하여 검색을 수행한다. 이 과정에서 클라이언트는 검색 방식의 구체적인 구현에 대해서는 알 필요가 없으며, 단지 searchByCategory 또는 searchByKeyword 메소드를 호출하기만 하면 된다.

|  |
| --- |
| **CategorySearch** |
|  |
| + search(String category): List<Product> |

|  |
| --- |
| ***SearchStrategy*** |
|  |
| *+ search(String parameter): List<Product>* |

|  |
| --- |
| **KeywordSearch** |
|  |
| + search(String keyword): List<Product> |

|  |
| --- |
| **ProductSearchEngine** |
|  |
| + searchByCategory(String category): List<Product>  + searchByKeyword(String keyword): List<Product>  - getSearchStrategy(String type): SearchStrategy |

**3. OrderManagementSystem - Observer (행동 패턴)**

- 옵저버 패턴은 객체의 상태 변화를 관찰하고, 변화가 있을 때마다 하나 이상의 의존 객체에 자동으로 알리는 것을 가능하게 하는 패턴이다.

- OrderManagementSystem에서 주문 상태의 변화(예: 주문 접수, 배송 시작)를 다른 시스템 컴포넌트(예: 재고 관리, 고객 통지 시스템)에 통지해야 할 때 유용하다.

|  |
| --- |
| **Order** |
| - orderID: String  - status: String |
| + setStatus(String status): void |

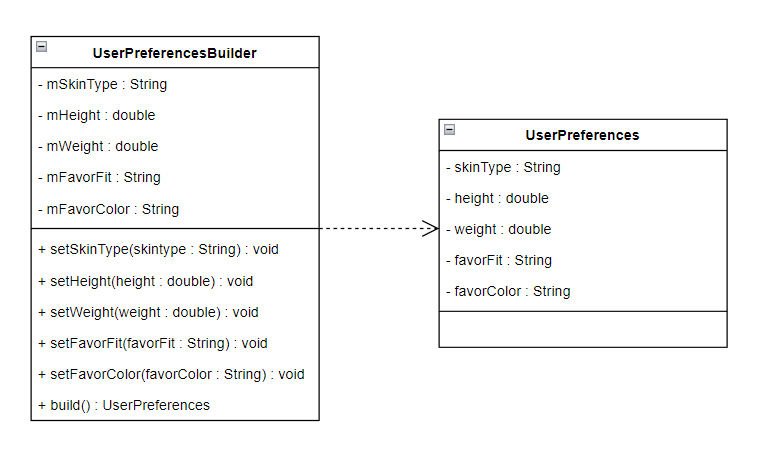
|  |
| --- |
| <<interface>>  **OrderObserver** |
|  |
| + update(Order order) |

orders

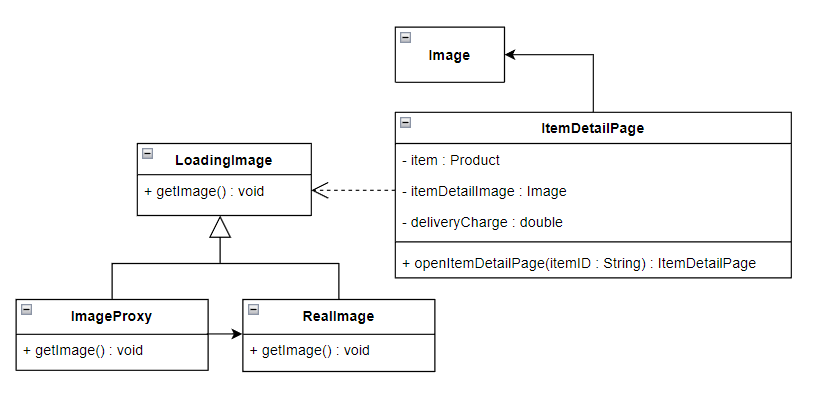
|  |
| --- |
| **OrderManagementSystem** |
| - orders: List<Order>  - observers: List<OrderObserver> |
| + addObserver(OrderObserver observer): void  + updateOrderStatus(Order order, String status): void  - notifyObservers(Order order): void |

1. **UserPreferences - Builder (생성 패턴)**

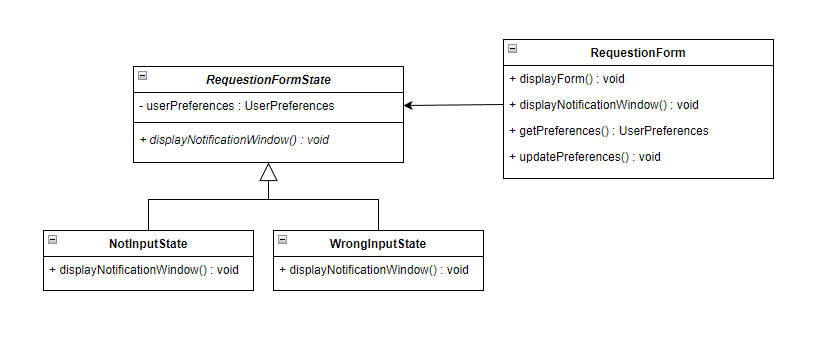
* Builder란 생성자가 여러 개라 객체 생성이 복잡할 경우, 생성자를 직접 호출하지 않고 Builder 클래스를 이용해 객체를 생성하는 디자인 패턴이다.
* UserPreferences는 코디 제안에서 사용자가 페이지에서 작성한 양식의 데이터를 담고 있는 객체인데, 해당 양식에는 여러 개의 필수 문항과 선택 문항이 존재하며, 선택 문항의 경우 데이터가 들어오지 않을 수 있다. 따라서 가진 멤버 변수가 많고, 선택 문항의 기입 여부에 따라 생성자를 여러 개 두어야 하는 UserPreferences에는 Builder 패턴이 적합하다.



1. **ItemDetailPage - Proxy (구조 패턴)**

* Proxy란 원래 객체의 대리 객체를 생성해야 할 필요가 있는 경우( ex : 다량의 객체를 불러와야 할 때 불러오기 완료 전 임시로 두는 객체 등) 사용하는 디자인 패턴이다.
* 상품 상세 이미지의 경우 스크롤을 많이 내려야 할 정도로 긴 이미지 또는 많은 이미지가 대다수이고, 페이지에서 가장 중요한 내용을 담고 있기 때문에 Proxy 패턴을 사용해 먼저 이미지가 로드 될 공간을 확보함과 동시에 사용자에게 이미지를 불러오고 있음을 시각적으로 표현한다.

1. **RequestionForm(displayNotificationWindow) - state (행동 패턴)**

* state란 객체의 내부 상태에 따라 객체의 행동 방식을 변경하고 싶을 때 state 추상 클래스를 생성한 후 다형성을 이용해 상태에 따라 다른 행동을 하도록 만드는 패턴이다.
* RequestionFrom의 displayNorificationWindow 메소드의 경우 양식 중 사용자가 표시하지 않은 필수 항목 또는 잘못 기입한 항목을 짚어 경고창(알림창)을 띄우려고 할 경우, state 패턴을 이용해 미기입 상황과 오기입 상황의 알림을 분리하여 사용할 수 있기에 적합하다.